pC*: Efficient and Portable Runtime Support for Data-Parallel Languages

(Ph.D. Dissertation)

Peter Alfred Bigot

UA CS TR-96-8 *and* ORC TR-91-1

Technical Report TR-96-8
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Technical Report ORC-TR-96-1
Oasis Research Center, Inc.

April 11, 1996

PC*: EFFICIENT AND PORTABLE RUNTIME SUPPORT FOR DATA-PARALLEL LANGUAGES

by

Peter Alfred Bigot

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 6

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ____________________________________

ACKNOWLEDGMENTS

The research described in this dissertation could have been neither undertaken nor completed without the assistance of a variety of people. In particular, the existence of the dissertation proper is the direct result of the willingness of my advisor, Saumya Debray, to support my sudden switch in research topics, from semantic and implementation issues in logic programming languages, to run-time systems for distributed data-parallel execution. Similar thanks are due to the other members of my committee, Todd Proebsting and Pete Downey. Such expressions of support when graduate students have an opportunity and desire to pursue independent research seem regrettably rare.

The work itself could not have been performed without the support of Oasis Research Center, and Charlie Turner in particular. Charlie provided me access to a variety of state-of-the-art computers which were capable of tackling large-scale problems without limitations such as tiny workstation clusters or inadequate computational power and memory—limitations which would have obscured the existence of many of the issues discussed in this work. The value of performing research in the context of real-world problems cannot be understated. Viability of the pC* system in a production environment was enhanced through discussions with Charlie Turner, Michael Pagels, David Izraelevitz, and Steve Swartz, all of ORC.

Thanks are also due to Phil Hatcher of the University of New Hampshire, who provided me with an early version of the UNH C* compiler, from which the work described in this dissertation evolved. Without this initial "leg up", the mundanities of developing a grammar for C* and otherwise getting to a working baseline system would have subtracted from time available to pursue the interesting research questions addressed herein.

Weekly lunches with Ed Menze, who has always been willing to quietly accept my off-the-wall quirks even when he didn't quite follow them himself, helped me preserve my sanity during a particularly trying period. Discussions with Barbara Hales on gender issues in cultural studies provided a welcome change from debugging communications interfaces.

The early years of my graduate experience were supported by an Office of Naval Research-funded National Defense Science and Engineering Graduate Fellowship, and by an AT&T Bell Laboratories Foundation PhD Fellowship. Support for the development of pC* was provided in part by the Advanced Research Projects Agency under U.S. Army Topographic Engineering Center contract DACA76-93-C-0026, and by Oasis Research Center under IRAD-7001-002.

To my parents

TABLE OF CONTENTS

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

LIST OF FIGURES

LIST OF FIGURES—*Continued*

LIST OF FIGURES—*Continued*

LIST OF TABLES

ABSTRACT

A variety of historically-proven computer languages have recently been extended to support parallel computation in a *data-parallel* framework. The performance capabilities of modern microprocessors have made the “cluster-of-workstations” model of parallel computing more attractive, by permitting organizations to network together workstations to solve problems in concert, without the need to buy specialized and expensive supercomputers or mainframes.

For the most part, research on these extended languages has focused on compile-time analyses which detect data dependencies and use user-provided hints to distribute data and encode the necessary communication operations between nodes in a multiprocessor system. These analyses have shown their value when the necessary hints are provided, but require more information at compile-time than may be available in large-scale real-world programs.

This dissertation focuses on elements important to an efficient and portable implementation of runtime support for data-parallel languages, to the near absence of any reliance on compile-time information. We consider issues ranging from data distribution and global/local address conversion, through a communication framework intended to support modern networked computers, and optimizations for a variety of communications patterns common to data-parallel programs. The discussion is grounded in a complete implementation of a data-parallel language, C*, on stock workstations connected with standard network hardware. The performance of the resulting system is evaluated on a set of eight benchmark programs by comparing it to optimized sequential solutions to the same problems, and to the reference implementation of C* on the Connection Machine CM5 supercomputer. Our implementation, denoted pC* for “portable C*”, generally performs within a factor of four of the optimized sequential algorithms. In addition, the optimizations developed in this dissertation permit a cluster of twelve workstations connected with Ethernet to outperform a sixty-four node CM5 in absolute performance on three of the eight benchmarks.

Though we specifically address the issues of runtime support for C*, the material in this dissertation applies equally well to a variety of other parallel systems, especially the data-parallel features of Fortran 90 and High Performance Fortran.

CHAPTER 1
SO WHAT'S THIS ALL ABOUT

> It seems disjointed and jumps around like water on a griddle, but it all comes together, so be patient.
>
> — Harlan Ellison, "Revealed at Last! What Killed the Dinosaurs! And You Don't Look So Terrific Yourself."

The calculation capability of computer systems has increased by orders of magnitude over their history, to the point where modern workstations, even personal notebook computers, have performances that match those of state-of-the-art mainframe computers of as little as ten years ago (Hennessy & Patterson, 1990). Unfortunately, this increase in the ability of single-processor computers to handle larger problem sizes has only whetted the appetite of scientists and researchers who are interested in problems that would require months or years of processing to solve. As software and hardware technology matures, interest and acceptance of parallel computation—linking many single processors together to solve a problem in concert—has increased.

Some software mechanism must be available for programmers to take advantage of these parallel hardware systems. Three primary techniques have been considered:

- Modify the compilation systems for standard sequential languages to detect opportunities for exploiting parallelism, and generate code which takes advantage of those opportunities;
- Develop new languages which have explicit constructs for parallel computation, such as distribution directives, synchronization primitives, and communication routines; and
- Extend current sequential languages with constructs which provide their compilers with information necessary to generate efficient code that runs on a parallel system.

The first two techniques received the most attention in early work on parallel systems. Previous data-dependence analyses developed to permit vectorization optimizations on vector-based supercomputers such as the Cray-2 could be extended to allow distribution of computation amongst separate processors (Banerjee, 1988). An ability to detect parallelism from previously-written "dusty-deck" programs would capitalize on a large investment in extant code. However, a consensus seems to be emerging that these techniques are inherently hampered by the fact that such codes were usually written with a particular system (hardware and/or software) in mind, and the opportunities for parallelization are often so deeply hidden that they cannot be effectively extracted (Adve, Carle, Granston, Hiranandani, Kennedy, Koelbel, Kremer, Mellor-Crummey, Warren, & Tseng, 1994).

The second method provides the programmer with an ability to control the computation completely, taking full advantage of the underlying parallel system. It is now acknowledged, though, that these parallel languages tend to result in programs that are hard to understand, difficult to debug, and often limited to a particular concept of a parallel system, such as uniform shared memory or distributed memory machines (Harris, Bircsak, Bolduc, Diewald, Gale, Johnson, Lee, Nelson, & Offner, 1995).

More recently, attention has turned to the third method. Proposed extensions to proven languages such as C (American National Standards Institute, 1989; Numerical C Extensions Group of X3J11, 1994) and Fortran (Adams, Brainerd, Martin, Smith, & Wagener, 1992; High Performance Fortran Forum, 1993) allow a programmer to express particular algorithms or operations in a high level form which the compiler can use to translate into code that implements the operations in an effective way on its target hardware. In addition, the backwards compatibility inherent from the underlying language means that investments in large software projects are not totally lost: core routines can be replaced with new code using the parallel features, while the less critical (and often larger) support code can be reused.

Two paradigms for how parallelism can be expressed are *data parallelism* (Hillis & Steele Jr., 1986) and *control parallelism*. The control-parallel paradigm, generally embodied in languages with explicit parallel constructs, requires the programmer to distribute work herself, and take responsibility for starting and synchronizing worker processes, and ensuring that each computing process is given the data it needs before it begins to work. In a data-parallel implementation, parallelism is implicit in the distribution of large data structures amongst many computing devices, each of which is responsible for performing operations on a subset of the entire problem. While a control parallel implementation provides fine control of the computation (in essence, an "assembly language" of parallel programming), data parallel systems are generally more easy to program, and are capable of expressing effective solutions to a wide variety of problems (Fox, 1988).

Two significant extensions for parallel programming support the data-parallel paradigm. The most recent standard for Fortran, denoted Fortran 90, contains support for operating on whole arrays without an enclosing `DO` loop: conceptually, the specified operation is performed on each element of the array simultaneously, and it is up to the Fortran 90 system to ensure that this is done correctly. Similar but more extensive support for data-parallel programming is in High Performance Fortran (HPF Forum, 1993). Many of the data parallel features in these languages are based on experience with an extension to C, called C* (pronounced "see-star"), developed at Thinking Machines Corporation (Frankel, 1991) to support their massively-parallel SIMD system, the CM2. C* has also birthed an alternative proposed data-parallel C, DPCE (Numerical C Extensions Group of X3J11, 1994). A variety of current research projects are investigating mechanisms that support these language extensions.

Optimizations for a parallel system can be implemented at three different levels:

- Control and dataflow analysis in the compiler can improve on the programmer's notations or detect special cases which can be handled more efficiently than a general case.

If communication patterns can be derived, part of the work involved in distributing data can be done at compile time, making the execution more efficient.

- The support system invoked at runtime, i.e. the mechanisms used to communicate values between nodes or implement control operations that are not resolvable at compile time, can be optimized for the types of operations expected in the language, or again to support special cases that can be detected either at compile time or during execution.

- At a lower level, the runtime system can be optimized for a particular host platform, by providing support within the operating system or hardware for fast communications. Research in this area also applies to general problems in networked computing.

While many research programs are devoted to compile time analyses (Tseng, 1993; Bozkus, Choudhary, Fox, Haupt, Ranka, & Wu, 1994), this approach can fail if the source program does not provide adequate information for the analyses to find opportunities to exploit parallelism.

In this dissertation, we focus on optimizations that can be performed solely at runtime, with no or minimal compiler support. We consider issues ranging from data distribution and runtime data structures, to a framework for portable inter-node message transmission and mechanisms for efficient communication of data between nodes with a variety of communication patterns. Because it is difficult to determine the synergistic effect of particular optimizations which are clearly desirable when only kernel computations are considered, we frame the approach described here in the context of a portable implementation of C* specifically intended to support a large extant image processing system developed at Oasis Research Center, which was originally developed to run on the Connection Machine CM5 system. The algorithms in this system are required to work on real-world programs and data sets, a constraint which has had a significant effect on the level of detail necessary to implement a complete and reliable runtime system. For example, image data such as that available from the Landsat satellite imaging system is often measured in gigabytes; a single multi-spectral Landsat image will be over 200MB (Richards, 1994). The algorithms often use a hierarchical view of data which narrows the focus of operations to a data-dependent subset of an image (Turner & Turner, 1994), which means that we will not know until runtime what size of data we will be working on in a particular routine, making compile-time analyses ineffective.

pC* ("portable C*") is a complete implementation of the C* language, designed to be fully compatible with C* as implemented on the Connection Machine CM5, from core language up to and including the complex computation-and-communication routines in the TMC `cscomm` library. The system was designed primarily to support stock workstations networked with standard interconnects such as Ethernet, while retaining a high degree of portability which we have proved by running the system on a variety of symmetric and distributed multiprocessors. We make certain basic assumptions about the type of interconnection mechanism supported, and show how these assumptions percolate through the entire system to allow a variety of optimizations in the runtime system and opportunities to easily support compile-time analyses. The intent of this dissertation is to support the following thesis:

> The performance of runtime support routines is fundamental to the performance of a parallel language/system: no compiler optimization can compensate for inefficient communications or memory operations. Carefully crafted runtime algorithms can yield adequate performance on their own, and independently provide support for compiler-assisted optimizations. Adequate performance can be achieved in a portable manner, though OS support for buffers and direct hardware control could improve this performance further.

We support this thesis in the following chapters.

Chapter 2 We describe the C* language, focusing on the features of particular significance to the runtime system: the paradigm with which data is viewed and the type of communications operations that are performed in programs, especially related to image processing. We go on to describe the genesis of the pC* system from the C* compiler developed at the University of New Hampshire by Phil Hatcher and his group, and describe its current status. We close with a description of how the material in this dissertation relates to and can support other data-parallel languages and systems, such as Fortran 90.

Chapter 3 In this chapter we examine in detail issues relating to data distribution, including runtime structures and the effect on performance of address translation between the programmer's global view and the runtime system's internal view. We show how translation techniques allow a cache sensitive data access pattern when straightforward non-contiguous access would result in very bad performance. We conclude with an examination of how uniform use of this access pattern permits an optimized encoding of a fundamental C* construct, context, in a way that reduces memory usage by up to 99%, and run time by as much as 25–50% on common programs.

Chapter 4 We consider the design of a communications framework in the context of the requirements of our system: correctness, portability, and efficiency. We outline a three-level hierarchy which isolates system-specific routines to a small set of well-defined and limited procedures, using a set of intermediate routines to handle issues of buffering and unrestricted communications in a machine-independent fashion, and supporting language-specific complex high-level communication operations. In the context of this framework, we examine a variety of algorithms to implement one-to-all broadcast and all-to-all reductions on both point-to-point and multicast protocols over Ethernet, and how the effect of imposing reliability requirements at user-level on top of an Ethernet-based UDP implementation can make use of hardware broadcast not as beneficial as one would normally expect.

Chapter 5 We use the intermediate routines of the framework described in the previous chapter to implement an efficient mechanism for moving data between nodes in an arbitrary fashion ("irregular communication" in the distributed-processing literature). We also examine a special type of communication pattern, common in certain image processing algorithms, and show how a heuristic can detect the pattern at runtime at little-to-no cost, and enable a variant communication implementation which can save up to 85% of the runtime by reducing communication volume.

Chapter 6 We examine how a simple mechanism can be used to emulate multi-dimensional loops at runtime without knowing the number of loops that will be needed. The mechanism is used to develop a method of quickly forming a common type of C* context using minimal compiler support. More importantly, it can be used to implement a general mechanism for grid-based communications, operating on data of arbitrary rank and dimension, with simultaneous shifts in multiple directions. The resulting general implementation is competitive with optimized implementations of special cases of grid communications, and is orders of magnitude faster than a straightforward general implementation.

Chapter 7 In this chapter we use a set of eight benchmark programs to measure the effectiveness of particular optimizations described in previous chapters, and the pC* system as a whole. We contrast the system on its native platform (twelve networked Sun SPARC-Stations connected with Ethernet) with optimized C solutions of the same algorithms, and a different portable C* implementation. We show portability by giving performance numbers on a symmetric multiprocessor (a Silicon Graphics 4D340), and the distributed memory Intel Paragon. We also compare the native platform performance of pC* with the performance of the reference implementation of C* on the Connection Machine CM5, showing that the optimizations in the previous chapters permit pC* to solve three of the eight benchmarks faster in real time than a sixty-four node, multi-million dollar supercomputer.

Chapter 8 We review the major contributions of the dissertation.

A hallmark of this work is its reliance on detailed experimental evaluation at all phases of development, to help build an understanding of the accuracy of our initial perceptions of an issue as important or unimportant. Within this dissertation, we often present initial and intermediate results along with our final decisions, so the reader has an opportunity to follow the development of our ideas and, we hope, build on her own intuitions. We express some code and data structures using a pseudo-code very similar to C (American National Standards Institute, 1989). These excerpts should not be taken as the only way to express an idea—in fact, the implementation in pC* usually differs for mundane reasons—but do often contain some nugget which has guided our path to a particular solution.

For the reader who is frustrated by our "whodunnit" approach, we provide a very short summary at the start of each chapter, which outlines its results. In conjunction with the index, the summaries can be used to jump to areas of particular interest. However, the bulk of the text is intended to address the question of "why" at least as much as it presents our conclusions of "what" and "how".

The core system described in this dissertation, as well as reports of subsequent research based on it, may be made available, at no charge, to interested researchers. To inquire about the current availability of pC*, send electronic mail to `pcstar-info@OasisRC.COM`.

CHAPTER 2

INTRODUCTION TO C* AND PC*

If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?
— Alan J. Perlis, Epigram #68

Before describing the details of the implementation of data parallel languages, it is necessary to understand the basic features of such a programming model. In this chapter we present the fundamental features of C (Thinking Machines Corporation, 1993; Frankel, 1991), one of the more mature data parallel languages, and include a discussion of the aspects of C* that make it difficult to use compiler-level optimizations for many real-world programs. We continue with a discussion of the genesis of the pC* system—the framework used in the remainder of the thesis—and briefly describe its current status. We conclude with a discussion of the relationship between C* and other data parallel languages, and the applicability of our work to these languages.*

2.1 Overview of C*

C* is a data-parallel language with extensions to ANSI C (American National Standards Institute, 1989), designed to support single-instruction multiple-data (SIMD) computers such as Thinking Machines Corporation's CM2. We will present here a very brief overview of the fundamental concepts of C*; this, in conjunction with more detailed examples throughout the text, should provide sufficient information for the reader to understand the issues involved in implementing the class of languages represented by C*. Readers interested in pursuing the syntax and semantics of the language in more depth are directed to (Frankel, 1991; Thinking Machines Corporation, 1993; Numerical C Extensions Group of X3J11, 1994).

2.1.1 Shape and Parallel Execution

Parallelism in C* is supported by extending C with *shape*s, which are similar to (multidimensional) arrays. Shapes are named objects, with *rank* or number of dimensions, and *dimension* or extent along each axis. Examples of shape declarations include:

```
shape [10] S1;
shape [20][30] S2;
```

`S1` is declared to be a one-dimensional shape with 10 elements or *position*s, while `S2` is a two-dimensional shape with a total of 600 positions. We use *dimension* both in the sense of the length of a particular axis, and in general to refer to the combined information of rank and all dimensions; the intended reading should be clear from context.

In C* terminology, a *scalar* type is any standard C data type, including aggregate types such as `structs`, but not arrays. A *parallel* type is a standard C data type augmented by a shape. Conceptually this denotes an object with a single scalar value at each position in the shape. For example, the declarations:

```
int:S1 i1, i1b;
float:S2 f2;
```

following the above shape declarations cause `i1` and `i1b` to be parallel integers—bundled groups of 10 integers treated as single objects—and `f2` to be a parallel float bundling 600 floats as a single object. We use "parallel variable" (or object or value, as appropriate, and often shortened to "pvar") to refer to the group of scalars together with their shape. We will generally use the term *position* to refer to an individual element in a shape, and *element* to refer to a particular individual scalar value in a parallel value; that is, position names an address, while element names a scalar value at an address. The distinction is not rigorous, however, and for prosaic convenience "element" will sometimes be used with the sense of "position".

C* is a *data-parallel* language, which means that parallelism is achieved by operating on program data in parallel. The standard C operators are extended to work on parallel objects with the expected semantics: each position in the shape is acted on separately and, conceptually, concurrently by different processors. We will use the notion of a *virtual processor* associated with each position to describe the semantics of C*. With the above declarations, the code fragment:

```
with (S1) {
  i1b = 2 * i1;
}
```

instructs each virtual processor to assign to the element of `i1b` that it owns twice the value of the element of `i1` that it owns. To ensure that such concurrent execution is well defined, C* uses the concept of *current shape*, introduced syntactically by the `with` construct used above, and requires that all parallel operands occurring in an expression be of the current shape. If a scalar value appears in a position which requires a parallel value, as with the constant 2 above, it is implicitly replicated in each virtual processor to satisfy the requirement. The keyword `current` yields a shape-valued expression which names the current shape, allowing access to shape information inside functions called within `with` bodies.

In addition to the source specification of shapes exhibited above, C* also permits shapes to be defined at runtime. Both the dimensions and the rank may be fixed at runtime, by calling a function which allocates a shape object. The code sequence:

```
shape S;
int * dims;
...
allocate_shape (&S, n, dims);
```

defines `S` to be an `n`-dimensional shape, where the extent of axis 0 is defined by `dims[0]`, axis 1 by `dims[1]`, etc. Following this, the new shape can be used in code just like any other:

```
with (S) {
  int:current iv;
  iv = ...
```

As a result, it may not be possible for the compiler to determine the number of positions of the shape, or even its rank, at the time the program is compiled. This is in stark contrast to languages such as Fortran, where at least the rank of a distributed array (analogous to a C* parallel variable) is known at compile time; most research compilers also assume full information about array bounds is available, if their most powerful techniques are to be applied (Koelbel, 1990; Hiranandani, Kennedy, & Tseng, 1993). Only recently has this begun to change (Agrawal, Sussman, & Saltz, 1995). Dynamic allocation of shapes is required for good performance in data-dependent algorithms, so assuming that full information is available at compile time leaves the problem of implementing these languages incompletely solved.

2.1.2 Communication and Position Addressing

As shapes are analogous to arrays, individual positions may be accessed using array indexing syntax. However, because a parallel object may be stored in a distributed fashion on a multi-computer, shape indexing is moved to the left side of the expression being indexed, to highlight that it may not have the same execution time profile as standard C indexing.

```
with (S1) {
  [4]i1 = 3;
}
```

This assigns the integer `3` to the fifth position of the shape (as with arrays, shape indexing begins with 0). Shape indexing is known as *left-indexing*, and left-indexing should always be assumed to have the potential to involve communication. Multidimensional left-indexing involves multiple bracketed indexing expressions, and the number of left-index expressions must exactly match the rank of the shape of the indexed expression.

If all index expressions are scalar, a single element of the parallel expression is named. Parallel index expressions must be parallel integers of the current shape. If any index expression is parallel, a communication operation is invoked. *Grid communication* results in shape **S**

iv	**0**	**1**	**2**	**3**
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

Figure 2.1: Results of `10*pcoord(0)+pcoord(1)` in a 4×4 shape

---

a transfer of data between elements linked by a fixed relative offset within the shape, while *general communication* transfers data between elements that may have no obvious structural relationship.

C* provides an intrinsic function[1] `pcoord(i)`, which evaluates to a parallel integer of the current shape whose values at each position contain the index along the `i`th axis of that position. For example, the following code assigns to `iv` the values depicted in figure 2.1[2]:

```
shape [4][4] S;
int:S iv, iv2;
with (S) {
  iv = 10 * pcoord (0) + pcoord (1);
}
```

Grid communication is invoked when the left index expressions consist of calls to `pcoord` on the corresponding axis, with a integral scalar offset. For simplicity, C* supports using the dot character as short-hand for the corresponding call to `pcoord` in this context. As an example, the expression:

```
iv2 = [.+1][.-1]iv;
```

assigns to a position $\langle i, j \rangle$ of `iv2` the value in position $\langle i+1, j-1 \rangle$ of `iv`, as depicted in figure 2.2. Use of a left-indexing communication expression as a C *rvalue* or plain value results in a *get* communication. Intuitively, each processor adds the necessary offsets to the indices which represent its own position, then requests the value from the named position. This results in a parallel value of current shape. Left-indexed communication can also be used in an *lvalue* context, i.e. as the target of an assignment, in which case a *send* communication is invoked: each virtual processor performs the index evaluation, then sends its value to the named position of the left-hand-side object. The send communication corresponding

1. Intrinsic functions—a notion borrowed from C++—are language constructs which are syntactically function calls but which may be recognized by the compiler and translated directly to any appropriate implementation. In addition to `pcoord`, C* counts the shape allocation functions among its intrinsics.

2. Throughout this work, rank-2 parallel variables will be shown with axis 0 proceeding vertically and axis 1 proceeding horizontally, from the upper left corner of the matrix.

shape **S**

iv2	0	1	2	3
0	?	10	11	12
1	?	20	21	22
2	?	30	31	32
3	?	?	?	?

Figure 2.2: Assignment `iv2 = [.+1][.-1]iv`

shape **S**

iv2	0	1	2	3
0	0	10	20	30
1	1	11	21	31
2	2	12	22	32
3	3	13	23	33

Figure 2.3: Assignment `iv2 = [pcoord(1)][pcoord(0)]iv`

---

to the above get communication is:

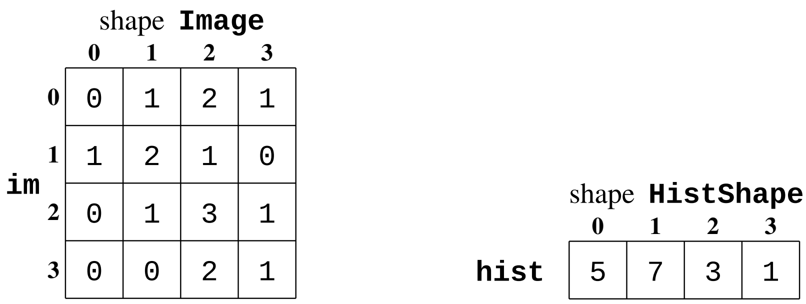
```
[.-1][.+1]iv2 = iv;
```

Note that the signs on the offsets have changed, as we are now providing the relative shift which yields the position to which our value is to be sent, rather than the position from which we wish to receive a value. Send and get operations are not exact duals: context (introduced in section 2.1.3) affects communication patterns differently for each, and send allows a combining communication (to be described later in this section).

General communication results when parallel left index expressions do not conform to the grid communication requirements: i.e., the index expression for axis i is not syntactically equivalent to `pcoord(`i`)+`c. In this case, the set of indices at a given position name the position from which this processor reads its value, or to which the processor sends its value. For example, matrix transposition may be implemented as follows, with results in figure 2.3:

```
iv2 = [pcoord(1)][pcoord(0)]iv;
```

Note that axis 0 has `pcoord(1)` as its index expression, so the processor which owns position $\langle i, j\rangle$ reads the value from processor $\langle j, i\rangle$. In a general communication all left indices must be parallel; if a scalar value is used as an index expression, it is implicitly extended to a parallel value just as is done in expression evaluation.



	0	1	2	3
0	0	1	2	1
1	1	2	1	0
2	0	1	3	1
3	0	0	2	1

	0	1	2	3
hist	5	7	3	1

Figure 2.4: Histogram example input and result

Send communications allow use of compound assignment, where the incoming values may be combined with the previous value in the target position and other incoming values to that position if the index expressions result in collisions. These features are exemplified by the idiom for image histograming: we have a shape whose elements represent pixels and have some integral data type, usually 8 or 16 bit, representing the brightness at that pixel. We wish to create a histogram which counts the number of occurrences of each intensity in the image. The following code does this, with image size and intensity range restricted so the illustrative example in figure 2.4 fits on the page:

```
shape [4][4] Image;
shape [4] HistShape;
int:Image im;
int:HistShape hist;

with (HistShape) hist = 0;
with (Image) {
  [im]hist += (int:current) 1;
}
```

In this example, the histogram target is initialized to zero. Then, while working in `Image` shape, each processor sends a one to the position in the histogram shape that is named by the image intensity at its own position. The compound assignment adds the incoming contributions, and the result is the number of positions in the image which have each intensity. This example also shows explicit casting of scalar values to parallel values, and the use of left indexing to convert from one shape to another.

2.1.3 Contextualization

The last major feature of C* that we will outline here is *contextualization*. In certain problems, there are regions of the shape where we do not want the processors to act; for example, window operations on images should not be performed at the edges where the window would have extended beyond the boundary of the shape. The observant reader will have noted in figure 2.2 that certain positions have undefined values. This is because those positions attempted to read from a shape position which did not exist. Reference to non-

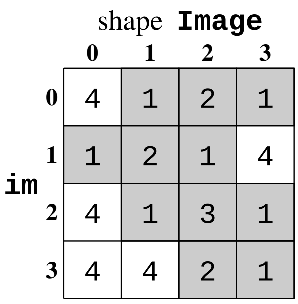


Figure 2.5: Contextualized assignment result

existent positions generally results in undefined behavior in C*: at best the system will quietly ignore the reference, but in some implementations other objects could be corrupted, just as may occur when accessing data outside array bounds in C.

Contextualization uses a conditional construct similar to sequential C's `if` statement, but using the keyword `where` and a parallel boolean expression to denote which positions are *active* for a block of code. For example, if we wished to replace all image pixels which have a 0 value with a different value, say 4, we could execute the following:

```
where (0 == im) {
  im = 4;
}
```

Each virtual processor will evaluate the conditional expression in the `where` statement. Only those which evaluate to a nonzero value will go on to execute the assignment in the body. If `im` initially has the value shown in figure 2.4, the value of `im` following the above contextualized assignment is shown in figure 2.5, where inactive positions are shaded. Note that the elements whose positions were inactive are unchanged. Careful programmers will avoid the undefined behavior mentioned for out-of-bounds accesses by protecting communication routines with contexts which will only reference valid locations. Thus, the grid communication shown in figure 2.2 should be coded as:

```
where ((dimof(current,0)-1 > pcoord (0)) &&
       (0 < pcoord (1))) {
  iv2 = [.+1][.-1]iv;
}
```

to ensure the index expressions do not stray outside axis bounds. The intrinsic function `dimof(s,i)` returns the scalar integer representing the dimension or extent of axis `i` in shape `s`.

Context is associated with the current shape, can be nested, and is persistent into functions called from within a `where` block: it is a dynamic feature of shapes. Therefore, in most cases the compiler is unable to determine the context under which a particular expression will be evaluated. If a new shape is entered with a `with` statement, the context will be reset to the context of the latest dynamically enclosing `where` block affecting that shape;

when the `with` statement is left, the previous shape is restored with its last known context. Like `if`, `where` statements have an optional `else` clause which is executed with the logical negation of the contextualization expression, subject to enclosing contexts. For example, with:

```
where (f1) {
  where (f2) {
    im = v1;
  } else {
    im = v2;
  }
}
```

the effect on `im` is:

- where both `f1` and `f2` are true `im` is assigned `v1`;
- where `f1` is true but `f2` is false `im` is assigned v2; and
- where `f1` is false `im` is unchanged.

The semantics of `where`/`else` is that both branches are executed in sequence, even if one might have no active positions. As such, any scalar operations that appear in either body (either directly or due to called functions) will be executed, and any side effects (scalar or parallel) in the `where` body will complete before operations in the `else` body begin. There is a corresponding `everywhere` statement which resets the context to completely active, since this cannot be represented by nested `where` statements.

2.1.4 Summary

We have introduced the major concepts of C*: parallelism is generated by augmenting a sequential language with *shape*s which represent groups of scalar objects as a single entity and which can be allocated and destroyed as execution progresses; data are moved about using *left-indexed* communication expressions; and evaluation can be restricted to certain elements through the use of *context*. The subset of C* outlined here is insufficient for serious programming: other C* features that will be addressed only in passing are:

- Reduction operators: these perform global operations to reduce all values in a shape down to a single scalar, such as the sum of elements in a parallel value
- More generic communication routines, which vary how addresses are specified, what the source value or destination object are, or permit fill values to be used when out-of-bounds accesses are performed
- Axis-specific computation routines, to perform reductions or spreads along axes in the shape independently (e.g., the sum of values in each column)

- Parallel prefix operations, to perform an associative operation along all elements in an axis, leaving intermediate results behind: e.g., the incremental sum along rows (the `scan` family of functions)

These functions and others are needed to allow C* to express many data parallel algorithms in various application domains, such as image processing and scientific computation. Although they have been implemented in pC*, the insight they offer is not as fundamental as the insights evoked by the core features described in more detail above, and as such this dissertation will focus on the issues of these core features, pointing out similarities in or additional requirements for the extended functions only in passing.

2.2 The pC* Implementation of C*

2.2.1 Genesis

This dissertation describes pC* ("portable C*"). The system was developed to support a group responsible for image processing software supporting a variety of applications in remote sensing (Richards, 1994), such as old growth forest mapping (Congalton, Green, & Teply, 1993), species habitat mapping (Turner & Turner, 1994; Aspinall & Veitch, 1993), land use change (Green, Kempka, & Lackey, 1994), and other instances where one wants to classify ground phenomena over wide areas quickly and inexpensively. The software in question was written in C* and had been running on Connection Machine supercomputers built by Thinking Machines Corporation. In the spring of 1994, the strong dependence of the system on TMC hardware and software was perceived as a serious weakness, especially given the questionable financial status of the company at the time: if TMC went out of business, the entire software system would need to be ported to a new platform, using a different language, with the concomitant loss of previous experience and libraries of algorithms. There was a strong interest in investigating the feasibility of an implementation of C* which would not be dependent on any particular hardware platform, allowing it to be moved from system to system based on whatever hardware was most cost-effective. A cluster-of-workstations model (Cheung & Reeves, 1992), using stock workstations connected with stock network hardware, was determined to be the best alternative to the previous "big iron" approach.

Since compatibility with TMC C* was of paramount importance, the choice of a replacement system was limited. The only available alternative that had a promise of robustness and would not suffer from the same single-source problems as TMC was a research implementation of C* developed at the University of New Hampshire by Phil Hatcher and others, based on previous work by Hatcher and Quinn on Dataparallel C (Hatcher & Quinn, 1991; Lapadula & Herold, 1994). We examined the UNH C* compiler and found that, although it implemented the core language, it had a variety of limitations which made it inefficient for large scale programming on the sort of data expected in image processing (data on the order of tens to hundreds of megabytes), and most importantly did not support any of the auxiliary C* functions that the image processing system required. Furthermore, the system was by design a research compiler, with focus primarily on compile-time analysis to improve communications behavior in core-language expression of mesh-based scientific algorithms, and some work on the Intel Delta network subsystem which had been its primary target, while more mundane issues needed for large programs and data but without real research value were discounted. What we needed was a solid C* implementation whose correctness and completeness were of significantly higher importance than speed, at least for the short-term. Therefore, we undertook to adapt the UNH system to our needs.

The original intent was simply to add the required additional functionality on top of the core UNH implementation. Over time, though, in support of the new functionality and reliability requirements, all components of the runtime system were replaced with new algorithms and data structures, and the front-end was extensively modified to use the changed runtime interface as well as programming features such as typedefs which are important in a production environment. The replacement algorithms were based on core assumptions about network features and getting as much speed and reliability as possible without relying on compiler analysis or limiting the source programs to restricted cases. Eventually it became clear that many of the algorithms and issues that had been addressed in the pC* system were of independent interest and had not been adequately addressed in the literature. The purpose of this dissertation is to examine the design and implementation of the resulting system. Before considering the details in the following chapters, we first examine the fundamental implementation model, retained from UNH C* and common to most implementations of distributed languages, then go on to present the current status of the system.

2.2.2 Basic Implementation Model

The fundamental implementation model of pC* is retained from the UNH C* system, and resembles that chosen for other distributed languages that translate to scalar languages, such as the original Dataparallel C (Hatcher & Quinn, 1991), Fortran-D/90D (Bozkus, Choudhary, Fox, Haupt, Ranka, & Wu, 1993; Choudhary, Fox, Hiranandani, Kennedy, Koelbel, Ranka, & Tseng, 1993) and SR (Andrews, Olsson, Coffin, Elshoff, Nilsen, Purdin, & Townsend, 1988). We use a single-program, multiple-data (SPMD) model, where each compute node in the system runs a copy of a common scalar program, operating on its own portion of the global data and using a library of message-passing routines to communicate with the other nodes in the computation. In the case of pC*, C* is translated to C code.

An important feature of the implementation model, and one which differs from SIMD-based implementations like the CM2, is that scalar data are replicated on all compute nodes, and all nodes perform the same operations on that scalar data. This allows us to handle Amdahl's observation that a portion of any program will be scalar computation, and hence not amenable to speedup, without suffering additional overhead by designating one compute node as responsible for performing scalar computation and distributing the results to other nodes. Since in C* all control flow at the program level is based on scalar values (using `if`, `while`, etc.), we can be assured that, unless somehow the same scalar expression evaluates to different values on different nodes,[3] control flow will be the same on all nodes, so incorrect behavior resulting from different execution paths will not occur. Synchronization is also easy to accommodate, since it is either implicit in communicating operations (such as reductions), or performed explicitly at the end of the communicating library routines: absent data flow analyses which would lift synchronization out of the library routines, no barriers need be inserted into the C translation.

Unlike other systems where communication behavior can generally be discovered by examining the program source, we do not attempt to generate calls to message passing routines directly, but rather call library routines to perform any operations associated with a communication (as is done in the Syracuse version of Fortran-90 (Choudhary *et al.*, 1993)). This results in smaller code, and isolates the opportunities for implementation errors and optimizations to one location rather than everywhere in a program that communication might occur. In trade, we may lose opportunities for latency-hiding communication/computation overlap when dataflow analysis is not performed. The decision is motivated primarily by the lack of information at compile time about shape dimension and distribution and the size of the cluster on which the program will run.

Each physical processor in the computation environment is responsible for a subset of the positions of each shape. Scalar computations in C*, including control flow, are translated directly to C, while regions of parallel code are grouped in the body of a loop which iterates over the positions of the shape held by the particular node: these loops are called *VP loops* (virtual-processor loops). Since communications invoke calls to library routines that operate on entire parallel values, they cannot appear within the body of VP loops. Thus communications and other library calls must be lifted out of—or split—VP loops, and the resulting parallel values stored in temporary variables by the compiler. For example, the C* code:

```
with (S) {
  iv2 = [.-1][.-1]iv + [.+1][.+1]iv;
}
```

would be translated to C code roughly comparable to:

```
readgrid (&tm1, iv, -1, -1);
readgrid (&tm2, iv,  1,  1);
for (vp = 0; vp < S.vplimit; vp++) {
  iv2 [vp] = tm1 [vp] + tm2 [vp];
}
```

(abstracting away from complexities not yet introduced). This does not take direct advantage of the fact that much of `tm1` and `tm2` are values that are available at positions $\mathtt{vp} + \delta_1$

3. Such behavior is contrary to the semantics of C*, but could occur with naïve implementations of reduction operations on floating point values, where the operations which are mathematically associative are not associative in implementation: a difference in order of evaluation on different nodes can result in different answers. Cf. section 4.3.2.

and $\mathtt{vp} + \delta_2$ respectively, because to do so would generally require knowledge about shape dimensions and distribution which is not available to the compiler. However, some of the techniques described later in this dissertation could be extended to permit direct reference to local data not present at the current position of the VP loop, at some (perhaps considerable) complication to the VP loop structure. These issues are reserved for future investigation.

2.2.3 Current Status

The current pC* system is a complete implementation of C*, including core language and auxiliary `cscomm` libraries. Though the primary development platform is networked Sun multiprocessors running Solaris 2.3 and using TCP sockets for communication, the system has been ported to and tested on Intel Pentiums running Linux and DEC Alphas running OSF/1 (single-process execution), the Portable Virtual Machine version 3 (PVM3), the System V message facility (under Solaris and Irix 5), a Sequent Symmetry using shared buffers to emulate a message passing architecture, and the Intel Paragon using the NX communications library. The system is currently used as the main development platform by four programmers in addition to the author. Over seventy thousand lines of code including several major image processing systems have been run using pC*; other applications such as shortest path and some core graph and linear algebra routines have also been implemented and used to solve problems.

The system has evolved from the UNH C* compiler of May 6, 1994, graciously provided to us by Phil Hatcher. The following significant changes to the front end translation program were made:

- Removed all dataflow analysis support. At the start of modifying the system to meet the pC* goals, it was deemed too difficult to guarantee reliability of the generated code when both the runtime library routines and dataflow analysis were required for correctness; over time, sufficient changes were made to the internal representations in the compiler and assumptions held by the runtime system that leaving such a large amount of unverified code in place was unwise from a software maintenance point-of-view. We do not feel this to be a great liability: evolution of the runtime library has resulted in a system which can integrate with a simple dataflow analysis with little effort, as will be described later in the dissertation. It seems unlikely, had we had the goal of supporting both fast runtime routines and compile-time dataflow analysis, that either could have been accomplished as well.

- The parsing system was overhauled, primarily to support maintaining C `typedefs` and ensuring compatibility of declarations between separately compiled modules.

- The analysis of `where` loops and corresponding generation of virtual-processor loops was enhanced to avoid overhead and generating unnecessary context maps (a similar optimization was independently added to a later version of the UNH C* compiler by the UNH researchers).

- Additional care was taken to ensure that compiler-allocated parallel values were collected when the blocks in which they were declared were exited "abnormally" (e.g., through `goto` or `break`). The image processing system generally works on shapes with millions of positions, making it very important to reclaim memory as quickly as possible.

The front-end generates C code, which is compiled and linked with a runtime library which implements memory management, communication, and the extended C* functions. The library and the runtime structures it uses have been completely redesigned several times by replacing core routines, until the library no longer bears any relation to the UNH C* implementation except in several hold-over utility functions for writing error messages. The details of the algorithms and structures used in the current runtime system comprise the bulk of this dissertation.

In addition to the compiler and runtime system, a distributed control system which allows execution of programs on a network of machines has been implemented. While some of its features are interesting in their own right, it is not significantly different from the control components of other systems such as PVM, and is not considered further here.

2.3 Related Parallel and Data-Parallel Systems

There are a variety of research systems which address the same general issues as this dissertation—language and implementation support for parallel programming—but none which seem to investigate runtime issues in data parallelism to the extent covered herein. The closest from the language point of view are the implementation of Dataparallel C (Hatcher & Quinn, 1991), based on the original version of the C* language (Rose & Steele Jr., 1987), and its successor C* compiler from the University of New Hampshire (Lapadula & Herold, 1994). The current literature on Dataparallel C seems to be addressed primarily to data distribution on networks of heterogeneous processors (Crandall & Quinn, 1993) and focuses on mathematical description of decomposition alternatives based on communication patterns rather than a detailed investigation of their implementation techniques and costs. Recent work on the University of New Hampshire C* compiler has addressed compiler analyses to improve communications behavior on irregular problems (Mason, Hatcher, & Chappelow, 1994). Neither system attempts to extract the level of performance that we demand from the runtime system alone.

Other languages provide a data parallel model of programming: most notable among these are the various extensions to Fortran which extend array semantics to operate over whole arrays at once, such as the array components of Fortran 90 (Adams *et al.*, 1992), Fortran-D (Tseng, 1993), and High Performance Fortran (HPF Forum, 1993). Research on the Fortran extensions tend to be more limited, though; since the Fortran paradigm has historically been one where the code expressed dependencies and explicitly coded loop bounds, research on data parallelism in Fortran has focused on analyses based on the assumption that the source code provides sufficient information to determine, for example, communication behavior or preferred data layout. This is in sharp contrast to C*, where most of the functionality used in image processing algorithms at least is buried deep inside library functions such as `scan` or specialized grid communication routines, where it is infeasible to specialize the code to a particular invocation. For example, the Fortran 90 equivalent to `where` does not reach down into user-defined functions called within its scope; hence, generation of contextualized loops is simplified considerably. Research compilers for data parallel Fortran (Tseng, 1993; Choudhary *et al.*, 1993) can have a sufficiently strong dependence on the availability of information at compile time that they are unable to translate general Fortran applications because run-time issues for the general case have not been addressed (Hiranandani *et al.*, 1993). The material in this dissertation, which contrarily focuses on run-time implementation to the near exclusion of available compile-time information, should complement these analysis techniques to result in a system which takes full advantage of all information available at all translation and execution stages.

APL (Gilman & Rose, 1984) is an array-based language which can be considered data-parallel. Translators from APL to C targeting both shared (Ching & Ju, 1991) and distributed (Ching & Katz, 1994) memory machines have been implemented. These implementations are geared towards “low-hanging fruit”—only array-parallel material (the contents of our VP loops) is considered. The reliance on shared memory and replicated data in these translators could be removed by merging the framework of this dissertation into the APL runtime system.

In addition, there are a variety of systems which use more explicit parallel constructs, but to which some of the data layout and access techniques described in this dissertation could be applied. Kali (Koelbel & Mehrotra, 1991) uses compile-time analyses to determine communication patterns and reduce communication overhead; the Kali implementation for runtime resolution of grid communication could benefit from the techniques of chapter 6. DINO (Rosing, Schnabel, & Weaver, 1991) requires explicit user specification of communications across nodes, hence does not embody a data-parallel model, but addresses the same sort of issues of data distribution and location resolution as we discuss in chapter 3. SPLIT-C (Culler, Dusseau, Goldstein, Krishnamurthy, Lumetta, von Eicken, & Yelick, 1993) is another explicitly parallel language which is designed to be portable and is being used for image processing applications (Fallah-Adl, JáJá, Liang, Kaufman, & Townshend, 1995). There have also been specialized languages such as Apply (Hamey, Webb, & Wu, 1989) which focus solely on neighborhood-based computations, and languages that permit nested rather than single-level data-parallelism (Blelloch, Chatterjee, Hardwick, Sipelstein, & Zagha, 1993).

C* has also served as the basis for an alternative data parallel version of C, as described in the Data Parallel C Extensions report (Numerical C Extensions Group of X3J11, 1994) of a subcommittee of X3J11, the American National Standards Institute committee responsible for the C programming language. DPCE is quite different from traditional C*: among other changes, it removes the need for current shape (requiring shape equivalence of operands instead), allows the user to specify slices out of a shape if interested in, say, only one row (something expressible in C* only through contortions and the use of context or general communications), adds parallel pointers to support irregular computations (C* does not permit pointers to appear inside parallel types), and adds nodal functions to allow the user to escape the data-parallel programming model (permitting each processor to execute different code up to a synchronization point). Though some of the features in DPCE are interesting, most such as nodal functions and parallel pointers have not yet been validated in a full-scale production compiler, and appear to result in some serious implementation difficulties unless strong restrictions are placed on internal representations. Due to their newness and other pragmatic restrictions outlined in section 2.2.1, the material described in this dissertation is addressed to C* as currently implemented by TMC supercomputers. Most of the core techniques should extend to a DPCE implementation without difficulty.

CHAPTER 3

IMPLEMENTATION OF PARALLEL VALUES AND CONTEXT

I see you stand like greyhounds in the slips,
Straining upon the start. The game's afoot: …
— Wm. Shakespeare, *King Henry V*, act III, scene *i*, line 31

We describe how pC distributes data amongst nodes in a computation, especially taking into account the costs of global/local address conversion, an issue which we argue has not been adequately addressed for the more complex distribution mechanisms described in the literature. We describe a mechanism for block distribution which allows the user a fairly fine control over placement of data, while still permitting very fast conversion between global and local addresses. We examine the issues of data access patterns on multidimensional objects, and measure the performance improvements of imposing a contiguous sequential access pattern on the data even when the operation being performed is conceptually non-contiguous. The techniques that permit this access pattern are those developed for address conversion. Uniform application of the contiguous access pattern permits a novel representation of context using run-length encoding, a method which yields space savings of up to 99% and time savings of 10–50% in common cases.*

Since computer programs operate on data, the operations required to access the data are perhaps the most important contributor to the efficiency of those programs. Along with classical issues of the appropriate structure for complex data, distributed multiprocessing raises the question of how data should be apportioned amongst processing nodes to balance the speedup from dividing the work equally with the slowdown of moving data to where they are needed, an operation that is generally quite expensive. This chapter examines the issues involved in choosing both data distribution and implementation data structures for parallel objects in C*. Since contextualization—restricting operations to apply to only certain data elements—is closely tied to data layout, we also address context representations that allow access to active positions and skip inactive positions with as little overhead as possible.

3.1 Issues in Data Distribution

A primary tenet of many research projects in parallel languages is that the fundamental determiner of performance is data distribution. The volume of literature on distribution of arrays in Fortran-like languages on a variety of hardware platforms is staggering

(Sheffler, Schreiber, Gilbert, & Chatterjee, 1994; Mahéo & Pazat, 1993; Knobe, Lukas, & Guy L. Steele, 1990). While we would not argue that data distribution is an insignificant contributor to performance, we feel that certain corollary issues have not been sufficiently addressed: specifically, the overhead involved in computing the owning node in schemes designed to distribute data equally over homogeneous or heterogeneous networks can be exorbitant, and any implementation that requires changing distribution between program points based on analysis of communication patterns leaves itself open to bad runtime behavior ("thrashing" of data distribution) if the analysis is inaccurate.

C* avoids some of these problems by dictating that a given shape has a fixed distribution, and all variables with that shape share the distribution. This ensures that operations that do not have communication coded explicitly by the programmer will occur without communication at runtime, presenting a predictable model that aids the programmer in understanding the performance of her code. Other problems are avoided by restricting the sorts of distribution that are supported in the system. While this restriction may affect certain algorithms by being unable to support the specialized distributions that are most appropriate for them, the limitation is outweighed by the corresponding improvement on other, more common, operations, where owner computation is simplified. The major distribution mechanisms and how they interact with C* programming are discussed below.

3.1.1 Data Distribution Options

If an algorithm requires no communication, then any distribution which doles out data in proportion to the computation power of each processing unit will result in a load-balanced system with maximal efficiency. However, most algorithms relate values at one position in a shape to others, either close or distant and with either relative or absolute position specifications. Two sources of overhead in these cases cut into the speedups from an equally-distributed system. The first is communication cost induced when the data needed for a computation reside on different processors, and the second is overhead induced by determining where the desired data live, using either absolute or relative indexing. As interconnection networks with higher bandwidth and lower latency become more common, the latter cost becomes increasingly more significant.

As an extreme case, it is possible to support a fully general distribution where there is no relation between the global address of an element (the set of indices that name that position in a shape) and its location at a particular address on a particular processor. While this is good from algorithmic and theoretical perspectives, in practice the need to do a complex computation or table lookup at each element to find its physical location will cause a drastic increase in the local processing costs of communication. We would like a mechanism where the hardest cases, e.g. general communication which has no structure, require very little computation per element, and easier cases such as structured grid communication are able to take advantage of their regularity and do address computation once for blocks of similar elements. In support of these desires, it is common to restrict distributions by supporting only a few regular layout options, and considering each axis separately (Tseng, 1993; HPF

Block0

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	0

Block1

	0	1	2	3	4	5	6	7
0	0	0	1	1	2	2	3	3
1	0	0	1	1	2	2	3	3
2	0	0	1	1	2	2	3	3
3	0	0	1	1	2	2	3	3

Block01

	0	1	2	3	4	5	6	7
0	0	0	0	0	1	1	1	1
1	0	0	0	0	1	1	1	1
2	2	2	2	2	3	3	3	3
3	2	2	2	2	3	3	3	3

Cyclic1

	0	1	2	3	4	5	6	7
0	0	1	2	3	0	1	2	3
1	0	1	2	3	0	1	2	3
2	0	1	2	3	0	1	2	3
3	0	1	2	3	0	1	2	3

Figure 3.1: Examples of Data Distributions. The block distributions are supported in pC*; the cyclic is not.

Forum, 1993).

Assuming a set of P processors (computation nodes) and a distributed one-dimensional shape (array) of N positions, the most common layout options are:

- Serial—all N positions are owned by one of the processors
- Block—the N positions are partitioned into P sequences, which are apportioned to each processor in turn
- Cyclic—the N positions are distributed in round-robin fashion over the P processors, with element i owned by processor $i \bmod P$

Cyclic and block can be combined into a *block-cyclic* distribution where chunks of size k are distributed in round-robin fashion. The issues are essentially the same as for cyclic distribution. Block distribution can require that each node has the same size partition as others—usually $\lfloor N/P \rfloor$, with any excess apportioned to the first $N \bmod P$ processors—or may allow arbitrarily sized subsequences. Some example distributions are shown in figure 3.1, with the values at each position naming the owning processor in a 4-node cluster, and heavier lines in the shape denoting borders between nodes.

The layout options supported in an implementation are fundamental to the performance of the system, and to its maintainability. There are several operations that are commonly performed and that depend on the distribution. These include:

- Given an element that is stored at offset o on processor p, what is its global coordinate along axis k of its shape?
- Given a vector of indices $\vec{i}$, what processor owns the named element, and what offset is it at on that processor?

- Given an element that is stored at offset o on processor p, what is the processor/offset pair of the element n positions away along axis k?

For simple distributions, such as equal-sized blocks along one axis, the computations required to do these conversions are straightforward and can be inlined directly. As an implementation or language comes to support more, and more complex, distributions, it becomes infeasible to duplicate the conversion code everywhere that it is required, and each such conversion requires a call to a function which determines the appropriate answer based on distribution information. This is especially an issue in a language like C*, where the shape dimension information is not available at compile time, or the conversions are being applied inside library routines where the distribution of the particular input cannot be determined *a priori*. Any system which attempts to generate code that does not rely on a hard-coded number of processors will encounter the same problems.

The above considerations rule out distributions more complex than the block/cyclic ones described above. Yet there is still a jump in complexity when supporting cyclic distributions, especially if full block-cyclic layout is allowed. It has been argued (Dongarra, van de Geijn, & Walker, 1992) that cyclic distributions are necessary for load balancing in certain linear algebra problems such as LU decomposition, where certain regions of data (rows or columns of a matrix) drop out as the computation progresses. In a naïve block-based implementation, processors will drop out one-by-one as rows become disabled in order, resulting in poor load balancing. Cyclic decomposition avoids this problem, because each row that drops out is owned by a different processor than the last row, so all processors remain active until the final rows are resolved.

However, even in this case, simple algorithmic changes permit block distributions to be probabilistically as effective as cyclic distributions. For example, if in LU decomposition we use virtual pivoting, where the chosen pivot row is not moved to its final position in the decomposition (an expensive operation in its own right) but instead the row number is recorded, load balance is effected by an assumption that pivot rows will be randomly distributed throughout the matrix. If this assumption should be invalid for expected inputs, or if a decomposition method is used which does not rely on pivoting for numerical accuracy (such as QR), a simple pre-processing step can effect a cyclic distribution at runtime under user control where necessary, without burdening all other computation by supporting a feature needed only for this case (see section 5.5.

Neighborhood-based grid computations, a primary component of image processing systems, also suffer under cyclic distributions, because strides which are not a multiple of the cycle length will induce communication for every position on the node, instead of having an internal block where all operations are local. The final straw that makes cyclic distribution untenable in C* is the existence of the parallel prefix `scan` functions: implementation of these, which requires saving the intermediate values as an operation is performed along an axis of a shape, becomes significantly more complex when values adjacent in the user's view of the data are never co-resident on the same processor.

3.1.2 Data Distribution in pC*

For these reasons, pC* supports only serial and block distributions, and meets the goal mentioned earlier of simple calculation for unstructured communication and amortizing the cost over similar sequences in grid calculations. However, the features of those distributions are not very restrictive, and application-specific load balancing and communication reduction are possible under the user's control. Furthermore, if cyclic distribution is truly required in a system which could use it more effectively, simple and common restrictions would permit many of the algorithms including the grid communications routines in chapter 6 to be extended to cover it as well.

If no other distribution is specified, the system will by default distribute axis 0 across the processors equally (using block distribution) while leaving higher axes undistributed (serial distribution). This allows the programmer to have an expected performance model for axial operations, regardless of the size of cluster the program will run on: i.e., she knows that "communication" along axis 1 or higher axes requires shifting data only within the processor, while operations along axis 0 will be more expensive. Should more control be required, the user is permitted to partition the axis in whatever fashion she desires, up to and including processors which receive empty sections, or allocating the entire extent to one processor. Distribution is also supported on any or all axes of a shape, under user control, subject to the restriction that the product of the number of blocks on all distributed axes yields the number of processors in the cluster: i.e., the distribution itself must yield an orthogonal partitioning of the shape. The sub-blocks of the partitioned shape are assigned to the processors in row-major order. Unlike data distribution on other parallel/distributed systems, neither the axis extent nor the sub-block size need be powers of 2, nor is there any benefit if they are. With the data sizes anticipated in an image processing system, rounding up to powers of 2 or otherwise requiring "nice" shape sizes can result in huge amounts of wasted space and concomitant performance loss (cf. the performance of matrix multiply on the CM5 in section 7.7).

As an example, figure 3.2 represents the distribution of a 8×8 shape across six processors, with axis 0 partitioned into three chunks of sizes 3, 1, and 4, and axis 1 partitioned into two chunks of sizes 2 and 6. The values in each cell name the processor number which owns that cell. This distribution mechanism, which allows partitioning of a shape into an arbitrary regular grid with *subgrid*s on each node, should be powerful enough to meet most load-balancing and communication-reducing needs. The details of the internal data structures and functions required to support it are presented in the next section.

3.2 Data Structures for Parallel Values in pC*

In this section we will examine the data structures used to implement parallel data in pC*, and the associated functions which perform the required conversions between internal representation and the indexing scheme used by the C* programmer. We will also address issues in allocating memory which uses these structures, and how to support access patterns shape **Image**

distvar

	0	1	2	3	4	5	6	7
0	0	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1
3	2	2	3	3	3	3	3	3
4	4	4	5	5	5	5	5	5
5	4	4	5	5	5	5	5	5
6	4	4	5	5	5	5	5	5
7	4	4	5	5	5	5	5	5

Figure 3.2: Example of Supported Block Distribution

which preserve good cache behavior while still computing values which require large strides between elements, as when summing the elements in columns of a matrix.

3.2.1 Implementation of `shape` in pC*

As noted in section 2.1, C* shapes encode multidimensional arrays with scalar values at each position. The arrays can be dynamically sized, not only in terms of the extent along each axis but also in terms of the number of axes. Although C* permits dynamic allocation of shapes, the user can specify when a shape is declared that the shape object can only hold shapes with a particular rank: for example, if parallel values with that shape are left-indexed, specifying that the shape will have two dimensions permits the compiler to make additional checks on the user's code. There are three classes of shape specification:

- A *fully unspecified* shape gives no information about either rank or extents. This shape may be used as the destination for any dynamically allocated shape. A fully-unspecified shape is declared in the following manner:

  ```
  /* Fully unspecified shape */
  shape UnspecShape;
  ```

  Note the absence of any left indexing.

- A *partially specified* shape provides information about its rank, but not its extent. This shape may be used as the destination for any dynamically allocated shape with the corresponding rank. A partially specified shape is declared by leaving the extents empty:

  ```
  /* Partially specified shape of 2 dimensions */
  shape [][]PartiallySpec2d;
  ```

```
typedef struct shape_base {
   int rank;              /* Rank of shape */
   int num_total;         /* Total number positions in the shape */
   int extent [MAX_RANK]; /* Length of each axis for whole shape */
   int ndistaxes;         /* Number axes w/non-serial distribution */
   axis_distribution_t disttype [MAX_RANK]; /* Axis distribution types */
   int distnblocks [MAX_RANK]; /* Number blocks each axis is split into */
   int * distbsizes [MAX_RANK]; /* Block sizes along each axis */
   int distpprod [MAX_RANK]; /* Axis-to-node support */
   PCS__ctx_rletype * context;  /* Pointer to current context */
   shape_pernode * dpernode; /* Distribution information per node */
} shape_base;
```

Figure 3.3: `shape_base` structure contents

- A *fully specified* shape provides both rank and extent information. This is the only case where (in conjunction with cluster size) the distribution can be calculated at compile time. Fully specified shapes are declared thusly:

  ```
  /* A 3-dimensional shape with 60 positions */
  shape [3][4][5] FullySpec3d;
  ```

  The extents, which are constant integral expressions, must be provided for all axes.

An unspecified or partially specified shape becomes fully specified when it is used as the destination of a dynamic shape allocation; it returns to its former specification level when the shape is deallocated. It is a runtime error to attempt to dynamically allocate into a shape object which is fully specified, either from declaration or dynamic allocation, or to allocate a rank k shape into partially specified shape of rank $j \neq k$. The two-stage shape data structure described here permits these checks to be performed.

A `shape` at user C* level is represented internally by a pointer to a `shape_base` structure, as defined in figure 3.3. A fully unspecified shape is represented by a null pointer; at allocation time, a base structure is dynamically allocated from system memory. A partially specified shape is represented by a pointer to a `shape_base` structure, but only some of the fields are filled in.

The fields in figure 3.3 contain the following information, which is defined only for fully specified shapes unless otherwise noted.

- `rank`: The number of dimensions in the shape. Must be a positive integer. Defined for partially or fully specified shapes.
- `num_total`: The total number of positions in the shape: the product of the extents.
- `extent[]`: The extent along each dimension. Must be positive integers.

```
typedef struct shape_pernode {
   int num_local;                 /* Number of local VPs */
   int localabove [PCS__MAX_RANK]; /* VPs above this node in each axis */
   int localdim [PCS__MAX_RANK];   /* VPs on this node in each axis */
   int num_per_axis [PCS__MAX_RANK]; /* Partial prods */
} shape_pernode;
```

Figure 3.4: `shape_pernode` structure contents

- `ndistaxes`: The number of axes which are distributed. Used for quick-lookup to see if communication is necessary with this shape, and for walking a shape in blocks as in chapter 6.
- `disttype[]`: For each axis, the distribution type: `serial` or `block`.
- `distnblocks[]`: For each axis, how many chunks is it split into?
- `distbsizes[][]`: For each axis, what are the sizes of the sections it is split into. Must be non-negative integers which sum, within each axis, to the extent of the axis.
- `distpprod[]`: A partial-product vector indexed by axis to aid in translating between processor number and shape partition subblock.
- `context *`: A pointer to the currently active context. This value is updated as context changes; see section 3.3.
- `dpernode []`: An array indexed by processor number to aid in determining the layout information for particular nodes.

Once a shape becomes fully specified, the dimension and distribution information is used to build information about the local layout on each node. An array of `shape_pernode` structures, one per processor, is allocated and assigned to the `dpernode` field of `shape_base`. The `shape_pernode` structure is defined in figure 3.4, with the following meanings per field:

- `num_local`: The number of elements that are stored on this node: the sub-grid size.
- `localabove[]`: For each axis, the position along the axis at which the data in this sub-grid starts.
- `localdim[]`: The extent along each axis for the subgrid that is held on this node.
- `num_per_axis[]`: Partial product information used for converting between local axis indices and the local offset.

```
Image->ndistaxes = 0;
Image->num_total = 1;
k = Image->rank - 1;
Image->distpprod[k] = 1;
Image->ndistaxes = (block_distribution == Image->disttype[k]);
Image->num_total = Image->extent[k];
while (0 <= --k) {
  Image->distpprod[k] = Image->distpprod[k+1];
  if (block_distribution == Image->disttype[k]) {
    Image->distpprod[k] *= Image->distnblocks[k];
    ++Image->ndistaxes;
  }
  Image->num_total *= Image->extent[k];
}
```

Figure 3.5: Calculation of Global Shape Geometry

Now let us consider in depth the process involved in generating the data used to convert between local (processor/offset pairs) and global (set of indices) addresses, using the distribution in figure 3.2 as an example. First, shape initialization assigns the basic fields thusly:[1]

```
Image->rank = 2;
Image->extent[0] = Image->extent[1] = 8;
Image->disttype[0] = block_distribution;
Image->distnblocks[0] = 3;
Image->distbsizes[0][0] = 3;
Image->distbsizes[0][1] = 1;
Image->distbsizes[0][2] = 4;
Image->disttype[1] = block_distribution;
Image->distnblocks[1] = 2;
Image->distbsizes[1][0] = 2;
Image->distbsizes[1][1] = 6;
```

With this information, the system can fill in `num_total` and `ndistaxes` trivially. Recall that we required that the product of the `distnblocks` fields for distributed axes yield the number of processors. This is so that the processor numbers, which range from 0 to $P-1$, can be mapped to coordinates in the partitioning of the shape, using the equation that processor p uses section s of the partitioning of axis k, where $s = (p/\texttt{distpprod}[k]) \bmod \texttt{distnblocks}[k]$. This provides a row-major mapping between processor numbers and partition coordinates. Code similar to that in figure 3.5 performs the necessary calculations, while information about local layout per-node is computed in the manner shown in figure 3.6. The resulting information for the distribution in figure 3.2 is

1. We give here only the effective code to perform the initialization for the example distribution. Initialization in the system is done through an allocation function which performs additional checks and allocates space for arrays such as `distbsizes`.

```
for (p = 0; p < P; p++) {
  pi = Image->pernode+p;
  pi->num_local = 1;
  k = Image->rank;
  while (0 <= --k) {
    pi->localabove[k] = 0;
    if (block_distribution == Image->disttype[k]) {
       sgi = (p / Image->distpprod[k]) % Image->distnblocks[k];
       pi->localdim[k] = Image->distbsizes [k][sgi];
       while (0 <= --sgi) {
         pi->localabove[k] += Image->distbsizes[k][sgi];
       }
    } else {
      pi->localdim[k] = Image->extent[k];
    }
    pi->num_per_axis [k] = pi->num_local;
    pi->num_local *= pi->localdim[k];
  }
}
```

Figure 3.6: Calculation of Local Shape Geometry

summarized in the tables in figure 3.7.

The internal representation of shapes is treated the same as user scalar values: the information is duplicated on all nodes, and all nodes possess information about the layout used on all other nodes. This simplifies the functions which convert between local and global information. We can now examine the implementation of the two primary conversion functions mentioned in section 3.1, leaving the third to chapter 6 where it is of most interest.

3.2.1.1 Internal Location to C* Index

Given an element that is stored at offset o on processor p, what is its global coordinate i along axis k of its shape?

This is the `pcoord` computation: the function that must be evaluated for each local offset o on a given processor p when generating the result of a call to `pcoord(k)`. The block distribution in conjunction with local row-major linearization permits a simple calculation using the partial product information in `num_per_axis`:

```
spn = shape->pernode + p;
i = spn->localabove [k] + (o/spn->num_per_axis[k]) % spn->localdim[k];
```

The division by `num_per_axis[k]` eliminates the effect of axes higher than k, while the modulo operation eliminates those below. This yields the position along axis k in the local subgrid; we need only add in the effect of prior elements along axis k stored on other nodes to get the final result.

This function is a good example of where the complexity of the distributions supported shape_base Data

Field	Value
rank	2
extent	$\langle 8,8\rangle$
num_total	64
ndistaxes	2
disttype	$\langle$block_distribution, block_distribution$\rangle$
distnblocks	3, 2
distbsizes	$\langle 3,1,4\rangle$, $\langle 2,6\rangle$
distpprod	$\langle 2,1\rangle$

shape_pernode Data

Field	$p=0$	$p=1$	$p=2$	$p=3$	$p=4$	$p=5$
num_local	6	18	2	6	8	24
localabove	$\langle 0,0\rangle$	$\langle 0,2\rangle$	$\langle 3,0\rangle$	$\langle 3,2\rangle$	$\langle 4,0\rangle$	$\langle 4,2\rangle$
localdim	$\langle 3,2\rangle$	$\langle 3,6\rangle$	$\langle 1,2\rangle$	$\langle 1,6\rangle$	$\langle 4,2\rangle$	$\langle 4,6\rangle$
num_per_axis	$\langle 2,1\rangle$	$\langle 6,1\rangle$	$\langle 2,1\rangle$	$\langle 6,1\rangle$	$\langle 2,1\rangle$	$\langle 6,1\rangle$

Figure 3.7: Shape data values for distribution in figure 3.2

in a system can have a strong effect on program performance, and it is worth spending some time examining the ways in which the implementation can be improved. `pcoord` generally appears in one of three places in a C* program, in decreasing order of frequency:

- In a grid communication, e.g. `[.-1][.+1]iv`. In this case, the implicit `pcoord` represented by `.` is not actually calculated in pC*; it is handled in the course of the grid communication, in the manner described in chapter 6.
- In a contextualization expression designed to prevent invalid positions from being accessed in grid operations; e.g.

```
where ((dimof(current,0)-1 > pcoord (0)) &&
       (0 < pcoord (1))) {
  iv2 = [.+1][.-1]iv;
}
```

  Again, in this case the `pcoord` is not calculated explicitly: the format of the `where` expression is noted by the compiler, and a context is built taking advantage of the regular form of the resulting boolean parallel expression (see sections 3.3 and 6.1).
- In some other fashion.

The third case is the only one where the `pcoord` calculation is actually performed with the above expression. In many such uses, `pcoord` will be called with a constant axis, as in the initialization expression for figure 2.1:

```
iv = 10 * pcoord(0) + pcoord (1);
```

Because the `pcoord` function is small, it can be defined in a C macro, with an expansion to the above expression in the resulting C code. The major time sinks in evaluating the expression are the two integer division operations (one divide, one modulo). Although most modern architectures implement these in hardware, they tend be several times slower than other instructions. By examining the construction of `num_per_axis` above and assuming that $o <$ `num_local` (guaranteed by the pC* code generator or checks in library functions), the reader will note that for $k = 0$, the modulo operation is unnecessary: the division already yields a value which is within the `localdim[0]` extent. Measurement within an old version of pC* indicated that evaluation of `pcoord(0)` was 50% slower when the unnecessary modulo operation was performed. On some small test programs this yielded a 10% slowdown overall, because of the large number of `pcoord` calls.[2] To improve performance we redefined the `pcoord` macro to check its axis and use an expression that does not perform the modulo operation when it is unnecessary:[3]

```
#define pcoord(_k,_o) (DimAbove(curshape,(_k)) + \
    ((0 == (_k)) \
       ? ((_o) / NPA(curshape,(_k))) \
       : (((_o) / NPA(curshape,(_k))) % DimLocal(curshape,(_k)))))
```

In many cases, the axis parameter `_k` is a constant at compile time, so the correct expression can be compiled without overhead. When the axis can be determined to be 0 the resulting expression is twice as fast as one which goes ahead and blindly does the modulo operation, even if the check must be performed at runtime; in the rare cases when the axis cannot be determined at compile time the check induces an overhead of only 5%. The effect in performance by such a small change to an already nearly trivial conversion calculation indicates the importance of keeping conversion functions as small and fast as possible.

However, we still have a `pcoord` function which contains at least one division operation. When `pcoord` is called in a VP loop which iterates through the processor's local offsets in sequence we can do much better by using a finite-state-machine implementation. Here the `pcoord` value cycles over the range of axis indices that are held on this node, incrementing once every so-many (`NPA(curshape,k)` to be exact) elements, and wrapping when it reaches the upper limit of the axis. Thus we can maintain a separate counter for the value of `pcoord(k)` with the following initialization before the VP loop:

```
pccnt = 0;
pcval = DimAbove (current, k);
```

2. Caveat: this was prior to the optimization which generates special context for the grid-bound protection expression above; in the current implementation many of these `pcoord` invocations would instead be routed to the boundary context code, which uses a different evaluation mechanism (cf. section 6.1).

3. We use `NPA` as a shorthand accessor function to the shape `num_per_axis` field; similarly for `DimAbove` and `DimLocal`.

pcoord type	constant 0	constant 1	variable 0	variable 1
div+mod, unopt	1176413	1176606	1176696	1179604
div+mod, opt	567973	1176518	630970	1218764
step/wrap	262926	263137	264576	362870
power-of-2, unopt	147764	147687	148041	147851
power-of-2, opt	147711	128411	169864	147656

Table 3.1: Evaluation of `pcoord` Implementation Alternatives. Times in µsec for 2^{20} conversions.

and the following increment step within the VP loop:

```
if (++pccnt == NPA (current, k)) {
  pccnt = 0;
  if ((DimAbove (current, k) + DimLocal (current, k)) == ++pcval) {
    pcval = DimAbove (current, k);
  }
}
```

This implementation, which can be considered a form of *strength reduction* (Aho, Sethi, & Ullman, 1986; Fischer & LeBlanc, Jr., 1988) performs much faster than the divide based mechanism. When it can be determined at compile-time that the axis in the `pcoord` is the highest axis in the shape (`k == rankof(current)-1`), we know that the distribution will guarantee that `NPA(current,k) == 1`, and the per-step increment can be reduced to:

```
if ((DimAbove (current, k) + DimLocal (current, k)) == ++pcval) {
  pcval = DimAbove (current, k);
}
```

This technique can be used to good effect anywhere in the library where it is necessary to walk the node's local data in sequence while retaining the pcoord values along a given axis; see section 3.2.4. The implementation here relies on a block-based distribution, though a technique similar to this or one based on the stride access pattern method of (Chatterjee, Gilbert, Long, Schreiber, & Teng, 1993) would be possible with a cyclic distribution. Irregular distributions are less likely to permit such a simple and fast `pcoord` implementation.

The performance of the various `pcoord` calculations in a C program designed to test alternative implementations is shown in the experimental results in table 3.1.[4] We consider the time requirements for various methods of pcoord calculation along each axis, where the axis is known at compile time (constant) or only at runtime (variable). Along with the unoptimized and optimized divide/modulo implementations and the step/wrap method described above, we present for comparison the time the `pcoord` calculation would take if the subgrid extents on axes 1 and higher were restricted to powers of 2. This has historically been done to improve exactly this sort of global-to-local address calculation, because it allows the di-

4. Timings run as the only active process on a 50MHz Sparc 20 using gcc 2.6.3 `-O2 -msupersparc`, on a rank 2 1024×1024 shape measuring time to generate `pcoord(k)` for various types of `k` and various implementations of `pcoord`. Values are in microseconds, and are the median of 5 runs.

vision and modulo operations to be performed with shifts and masks instead. The resulting functions are approximately 8 times faster than the unoptimized divide-based implementations on the tested hardware, though only 2-3 times faster than the step/wrap method, and it is straightforward to extend the `shape_pernode` structure to contain the necessary shift and mask values for each axis. However, the concomitant subgrid extent restriction is onerous: if the user wishes to extract an $n \times m$ block from an image for more detailed processing, she or the system must round m up to a power of 2. This may make the shape nearly twice as large as the region of interest, and requires extra care to either avoid processing the fill elements or ensure that the operations performed will not fail due to invalid data in the fill area. The step/wrap method is less than twice as slow as the shift/mask version except on the highest axis when this is not detected at compile-time, and has no restrictions on the axis extents. As such, limiting shape extents to powers-of-2 does not appear to be worthwhile, and pC* uses the step/wrap method for all pcoord operations within VP loops.

Outside VP loops the internal-location-to-C*-address conversion is used in several library functions. When we are operating on a single offset we always want the indices for all axes (e.g., to yield a global address for a particular position), and in this case, the `pcoord` computation can be placed in a loop which calculates the indices by peeling them out of the offset one-by-one; the step/wrap method is inappropriate in this case, and we must resort to using the division operations (one per axis). There is one obscure library function (`copy_multispread`) which requires computing all indices for each element in turn; it uses a variant of the multidimensional `for`-loop described in chapter 6.

3.2.1.2 C* Address to Internal Location

Given a set of indices `idx[]`*, what processor owns the named element, and what offset is it at on that processor?*

The inverse operation of mapping a user-provided C* global address to a processor/offset pair is less common, occurring most often when addressing an element of a parallel value as a scalar value through scalar left index expressions, as is done at each position when performing general communications (cf. chapter 5). It is somewhat more complex than the pcoord calculation above, requiring several loops to search for the owning processor based on the index values, and calculate the linear offset given the local indices within

Rank	Inline	Outline
1	0.4002	0.5802
2	0.5603	0.7402
3	0.7203	0.9003
4	0.8802	1.0604

Table 3.2: Per-Position Costs of C* to Local Index Conversion. Time in seconds for 10^6 conversions.

the subgrid stored on that processor.

```
pn = 0;
for (k = 0; k < shape->rank; k++) {
  d = idx [k];
  i = 0;
  while ((i < shape->distnblocks[k]) &&
         (d >= shape->distbsizes[k][i])) {
    d -= shape->distbsizes [k][i];
    i++;
  }
  ldims [k] = d;
  if (block == shape->disttype [k]) {
    pn += i*shape->distpprod[k];
  }
}
offs = 0;
for (k = 0; k < shape->rank; k++) {
  offs += ldims [k] * shape->pernode [pn]->num_per_axis [k];
}
```

Were this functionality required in many locations, it would be undesirable (for code bloat reasons) to generate it using a macro or by explicitly coding it, and it should be placed inside a function to ensure maintainability of the system. However, since there are relatively few locations where the conversion must be done and all are inside library routines (unlike `pcoord`, which can appear arbitrarily many times in converted user code), the code sequence can be encapsulated in a macro or inlined function to save the (rather significant) function call overhead on each position.

The cost of this function is given for shapes with ranks 1 to 4 in table 3.2, running on a 50MHz Sun Sparc 20. The test converts one million global index vectors to processor/offset pairs using the above code fragment. The time in the table indicates the average per-position cost in microseconds, comparing a method which inlined the conversion in the VP loop with one which placed it in a function which was called once for each position. The constant difference of 0.18sec is about what we would expect for one million function calls on a 50 MHz SS20 (measured independently to be approximately 166 nsec per call). This amounts to roughly 30% of the conversion time for a 2d shape, so it is clear that inlining is desirable.

To understand the performance with respect to communications overhead, an Ethernet packet of 1400 bytes[5] will hold 175 elements in a general communication of integers (nodes must receive both the value and its destination offset; see chapter 5). We can estimate the one-way transmission using the reliable TCP package of figure 4.9 on 10Mbps Ethernet to be approximately 750µsec for startup (half the bi-directional TCP overhead); we will assume that, since sending a packet does not require waiting for it to go across the wire, 1msec is a conservative (over-)estimate of the time required to transmit a 1400-byte message without waiting for any response (as is appropriate in this case).[6] The location calculation time using a rank-2 shape with the inlined conversion for the elements which go into that packet is 98µsec, approximately 10% of the per-packet transmission cost. The per-packet communications cost can be decreased significantly with network optimizations such as those outlined in section 4.5, increasing the proportion of time spent on local address computations. This again leads us to remark on the importance of ensuring that translations between global and internal addressing be as simple and fast as possible.

3.2.2 Runtime Memory Management

We must pause the discussion of implementation data structures to address a related issue—that of allocating space for them at runtime. Although shape structures require a certain amount of dynamic memory (e.g., allocating the per-node array once the cluster size is known), the amount of memory is fairly small. This is not the case for parallel variables and contexts. C* variable declarations have the same storage-classes as C declarations: ones at file scope or with `static` class persist throughout the life of the program, while `auto`-class variables inside blocks are created and destroyed when the block is entered and exited. Since the amount of memory required for a parallel variable is not known until runtime, and is often extremely large, it is not feasible to create pvars on the C stack. Therefore, we need a dynamic memory allocation scheme which allows easy creation and reclamation of parallel objects.

While many parallel objects are created in direct response to user declarations, some must be created by the compiler, for example to use as temporary values to hold the results of communications. A complex garbage collection scheme is not required, but it is necessary to reclaim, at the end of the block, both the user- and compiler-generated parallel variables which were allocated therein. A simple high-water mark collection scheme suits this well: the memory allocator must support marking a current allocation level when a block is entered, and reclaiming everything allocated since that point when it is left.

The implementation in pC* assigns allocations to the classes listed in figure 3.8. Allocation is performed on each node through a function which is given the size of the block required and a memory class with which it is associated. The memory is allocated using C's

5. See footnote 9 on page 90 for why sizes were limited to 1400 bytes.

6. It is difficult to determine the exact cost, since it depends on the status of system buffers, but simple timing tests indicated that absent flow control problems the `write(2)` system call for 1400 bytes to a TCP socket takes less than 500µsec on the experimental hardware. We will accept 1msec as an estimate to include the extra processing time that is required per chapter 4, though discussion in section 5.5 indicates a more complex estimation may be required for large data transfers.

```
typedef enum PCSRTMemClass {
   PCS__RTMC_CompilerTemp,      /* For compiler temporaries */
   PCS__RTMC_UserMem,           /* User pvars */
   PCS__RTMC_StackedContext,    /* Stacked contexts within where blocks */
   PCS__RTMC_ParFuncRetval,     /* Return values from parallel ftns */
   PCS__RTMC_ShapeDecl,         /* Non-dynamic shape information */
   PCS__RTMC_StaticPvar         /* Static local pvar */
} PCSRTMemClass;
```

Figure 3.8: Dynamic Memory Classes

`malloc` function, and a pointer to the block is saved in an array which emulates a stack of allocations in that class. High-water marks are indexes into the arrays, and memory is reclaimed in one pass by walking the array from the mark to the last allocation, calling `free` on each block to be freed. We chose to use `malloc` and `free` directly rather than cache blocks inside the pC* memory handler because the host system's allocation routines will perform necessary block splitting and merging as memory allocation patterns change, while duplicating this work inside the pC* memory handler (as was done in (Lapadula & Herold, 1994)) would be complex and liable to leave blocks of memory allocated but unused, especially when using dynamically allocated shapes of various sizes over a long program execution. A necessary step to ensure that memory use does not grow through leaks is to execute the reclamation code everywhere a block can be exited: not only when it is closed, but at `goto`, `break`, and `continue` statements as well. While the memory would be reclaimed when the enclosing block was left, a loop with a body in which reclamation code is consistently skipped due to non-structured resumption could easily consume all available memory.

The shape of file scope parallel variables must be fully specified at compile time, and the compiler generates initialization functions that are called prior to invoking the user's `main` function to do the necessary allocation in appropriate classes, as described in (Lapadula & Herold, 1994). Management of non-persistent allocations is performed on block entry and exit, using the high-water stack method outline above. Five of the six classes in figure 3.8 use this stack-based allocation: compiler temporaries, user pvars (with C `auto` storage class, i.e. declared inside blocks), stacked contexts (contexts allocated when a `where` block is entered), temporaries which hold parallel return values, and the space required for partially and fully-specified shape declarations inside blocks all can be reclaimed when the scope in which they were defined has been left.[7] The last class is used for values with `static` storage class (which are allocated and initialized the first time the block is entered, and must be of a shape which is fully specified at compile-time). Though high-water mark and reclaim is not used in this case, we collect the blocks used for static allocations so they can be freed when the program exits. Similarly, the memory allocated through the initializa-

7. Though all five are stack based, we do not attempt to coalesce them into a single allocation stack, because they are conceptually separate entities. Allocation and reclamation patterns can differ slightly between the classes, and there would be no benefit in memory use or code clarity by combining the classes.

```
typedef struct PCS__Pvar {
   PCS__shape_base * shape; /* Shape by which data is to be interpreted */
   char *base;              /* Base of data region for variable */
   int stride;              /* Bytes between subsequent elements */
} PCS__Pvar;

typedef struct PCS__PvarPtr {
   PCS__Pvar var;           /* Basic information about the variable */
   char * data;             /* Base of the data pointed to */
} PCS__PvarPtr;
```

Figure 3.9: Structures for Parallel Variables

tion functions for file-scope variables is freed by the compiler's exit function (by reclaiming the whole stack of blocks), leaving allocated only memory which the user has asked for specifically. Memory allocations in response to user commands such as `palloc` (for creating parallel variables of a given shape with given element size) or `allocate_shape` (for dynamic shape specification) are done using direct calls to `malloc`, since there is no need to be able to recognize them or treat them as distinct classes, and it is the user's responsibility to make the corresponding calls to `pfree` and `deallocate_shape`. Careful management of compiler-allocated memory permits development tools such as Purify (Pure Software, 1994), which analyzes memory usage and detects illegal accesses, to be used on the resulting C* programs, aiding in making the user's code robust by ensuring that access errors and leaked memory are due to problems in the user's code and are not caused or masked by similar problems in the runtime system.

3.2.3 Implementation of Parallel Variables

The data structures used for parallel variables are significantly simpler than those for shapes, and are given in figure 3.9. These are essentially the same as in (Lapadula & Herold, 1994), except for one change—relocating the information from a header which appears before the raw data to a structure dissociated with the data—which is required to support aliasing parallel variables between shapes, a feature used in a common image analysis technique described in an extended example below.

The basic variable structure contains a pointer to a `shape_base` structure, indicating the shape of the variable; a pointer to a data region of

```
spn->pernode[mynode].num_local
```

elements treated as a linear sequence; and a *stride* which gives the element size in bytes. A *pointer-to-parallel* consists of a parallel variable along with a separate pointer into the data region. This pointer will be in the range $[\texttt{var.base}, \texttt{var.base} + \texttt{var.stride})$, and permits the user to address specific fields of parallel aggregate types, such as structures or arrays. A pointer-to-parallel does not require a separate stride field, as it will use the stride of

```
char * PCS__t0, * PCS__t1;
int PCS__s0, PCS__s1;

PCS__t0 = iv.base;
PCS__s0 = iv.stride;
PCS__t1 = iv2.base;
PCS__s1 = iv2.stride;
PCS__vpi = 0;
PCS__vplimit = PCS__current->pernode[PCS__nodenum].num_local;
while (PCS__vpi < PCS__vplimit) {
  * (int *)PCS__t1 = 2 * *(int *)PCS__t0;
  PCS__t0 += PCS__s0;
  PCS__t1 += PCS__s1;
  PCS__vpi++;
}
```

Figure 3.10: C Translation of C* code `iv2 = 2 * iv`

---

the base variable. Pointers-to-parallel are valuable both by providing a way to dynamically allocate parallel variables through `palloc`, and by providing a way to pass parallel variables to functions by reference. This avoids the need to make a copy of an arbitrarily large block of data on each parallel function call, since C* retains C's call-by-value semantics even for parallel parameters.

Access to the elements of a parallel expression is made within virtual-processor loops by associating a C `char *` pointer with each parallel object, dereferencing a cast of that pointer to the appropriate type, then adding a stride to point to the next element. For example, the C* code

```
iv2 = 2 * iv;
```

would be translated into something like that given in figure 3.10. In cases where the parallel value has a scalar type and it is known that the stride is equal to the size of the type, the pointer dereferencing can be converted into array indexing; e.g. `PCS__t0 [PCS__vpi]`.

A more extended example will highlight the use of parallel variable structures and pointers, the need to be able to refer to parallel variables by different shapes, and the rationale for supporting arbitrarily-sized blocks in distribution specifications.

3.2.3.1 Extended Example: Shape Aliasing

Image analysis tends to operate on extremely large sets of data: for example, the data for a single patch of Landsat Thematic Mapper imagery covers a ground area 185 kilometers on each side with an 8-bit value for each of six frequency bands at each 30m $\times$ 30m pixel, yielding a 216MB data set (Richards, 1994). In many cases, a quick-and-dirty analysis can determine that there are only certain regions of the image that hold data of interest; in many of those cases, the structural relationship between the pixels in the regions is less important

elev (shape **Image**)

	0	1	2	3
0	4	1	2	1
1	1	2	1	4
2	4	1	3	1
3	4	4	2	1

imband1 (shape **Image**)

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

imband2 (shape **Image**)

	0	1	2	3
0	16	15	14	13
1	15	14	13	12
2	14	13	12	11
3	13	12	11	10

Figure 3.11: Shape Alias Example Data

than the values of features associated with them. For example, a pass over an image looking at elevation data could recognize pixels above a certain elevation, and perform a more detailed multi-band spectral analysis on those pixels to identify the type of ground cover in those regions (see, for example, (Turner & Turner, 1994)). Though context might restrict operations to the pixels of interest, thus saving time, we would still need to allocate space for the uninteresting pixels as long as we operated in the image shape. Therefore, we want to extract those pixels from the image into a smaller shape (Voorhees & Tucker, 1992).

As an example, consider the data in figure 3.11 as representing elevation data (`elev`) and two bands of spectral data that can be used for ground cover classification (`imband1` and `imband2`). We wish to extract the band data corresponding to pixels whose elevation is highest, and put those into new parallel variables to operate on. The code in figure 3.12 performs this function.[8] To further illustrate aspects of data distribution and shape aliasing, we have assumed a layout for the image data which distributes it along both rows and columns, on a 4-node cluster, with bold lines indicating cross-processor boundaries in the shapes. Let us now examine in detail what the code does.

- First, we create a new shape which is one-dimensional, has as many elements as `Image`, uses block distribution into as many sections as there are nodes in the cluster, and uses as the section sizes the `Image` subgrid sizes on each node. This ensures that we have a shape which has the same number of elements on each node as `Image`, but is one-dimensional.

- Next, we create two pointers-to-parallel which point to the same memory as the two-dimensional parallel variables `elev` and `imidx` but will treat that memory is though it were of shape `ImageIn1d`. The elevation data interpreted as a one-dimensional pvar is shown in figure 3.13. Note that the one-dimensional view is not simply a row-major flattening of the two-dimensional view, because we are retaining node ownership of the

8. The interface to detailed allocation in pC* is actually somewhat more complex than that shown in figure 3.12, but the differences would serve only to confuse the issue. As a policy decision, we retain TMC C* names for functions which perform the same operation as TMC versions or like `allocate_detailed_shape` are documented to be implementation-specific, while changing the names for functions which do not appear in TMC C*: hence the use of `PCS_shape_subgrid_sizes`, which is specific to pC*, but `CMC_change_pointer_shape`, which is supported in both implementations.

```
with (Image) everywhere {
  shape ImageIn1d;              /* Shape for 1d image */
  shape ReducedImage;           /* Shape for reduced data set */
  int:Image imidx;              /* Reduction conversion map */
  Peltype:ImageIn1d * elev1d;   /* Elevation data as 1d pvar */
  int:ImageIn1d * imidx1d;      /* Index as 1d pvar */

  /* Allocate a 1d shape with same subgrid sizes as 2d Image */
  allocate_detailed_shape (&ImageIn1d, 1, positionsof(Image),
     PCS__block_distribution, dimof(physical),
     PCS_shape_subgrid_sizes (Image));

  /* Alias both the image and the idx vector to 1d */
  elev1d = CMC_change_pointer_shape (&elev, ImageIn1d);
  imidx1d = CMC_change_pointer_shape (&imidx, ImageIn1d);
  imidx = -1;
  with (ImageIn1d) where (4 == *elev1d) {
    /* In 1d space, generate an enumeration of high-elevation
     * pixels */
    *imidx1d = enumerate (0, CMC_upward, CMC_exclusive, CMC_none,
      CMC_no_field);
    /* Determine the number of pixels which are turned on. */
    npix = += (int:current) 1;
  }
  deallocate_shape (&ImageIn1d);

  /* Allocate a new shape to hold the reduced image */
  allocate_shape (&ReducedImage, 1, npix);
  with (ReducedImage) {
    Peltype:current rband1, rband2;

    with (Image) where (0 <= imidx) {
      /* Send the band data for the high elevation pixels into
       * the pvars for the working shape */
      [imidx]rband1 = band1;
      [imidx]rband2 = band2;
    }
    /* Operate on the reduced image data here */
    ...
  }
  deallocate_shape (&ReducedImage);
}
```

Figure 3.12: Code for Shape Aliasing Example

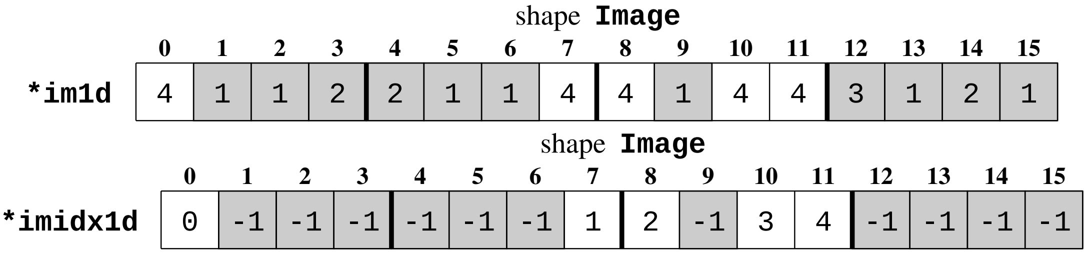


Figure 3.13: Elevation and Index Data in One Dimension

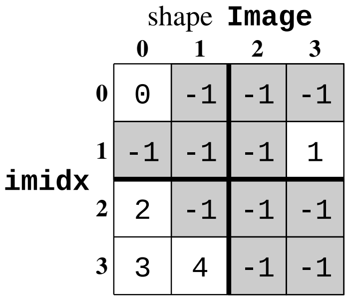


Figure 3.14: Reduction Index Variable

data: although the right half of the first row is owned by processor 1, the data appear following the left half of the second row, which is owned by processor 0.

- Now we can create the index variable. We initialize it to -1, so positions which do not satisfy the elevation test are recognizable, then switch to the one-dimensional shape and immediately restrict the context to the positions where the elevation is maximal. The inactive positions are shaded in figure 3.13.

- Under this restricted context, we call a C* auxiliary function `enumerate` which labels the active positions with the number of positions prior to them: this gives each high-elevation pixel a unique index. The result, viewed as a one-dimensional pvar, is also shown in figure 3.13. We determine how many positions pass the elevation test: this will be the required size of the shape for the reduced data.

- We can now free the one-dimensional image shape, since we are finished, and allocate a new one-dimensional shape to hold the reduced data.

- Next, we enter the reduced data shape, and declare variables of that shape for the band data. To send the band data to the new shape, we drop back into `Image` shape, restrict the context to positions which have valid indices, then send the image band data into the reduced shape pvars. The `imidx` pvar interpreted in `Image` space is in figure 3.14, and the band data in its final format are in figure 3.15.

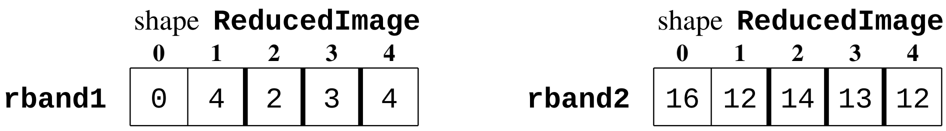


Figure 3.15: Reduced Band Data

Through the use of pvar aliasing, we are able to perform all the steps up to the final redistribution of the data without communication, except for a small amount within `enumerate`. After execution of the alias and distribution code, we have reduced the problem size to exactly what we need to work with, and have redistributed the necessary data over the entire cluster, eliminating any load imbalance due to an unequal distribution of high-elevation points in the original image.

This example illustrates some of the functions we wish to perform and how the internal representation supports them. The need to view an object as two-dimensional and one-dimensional data simultaneously comes from the object's natural form as a representation of an image area, conflicting with the desire to label a subset of the area with a linear sequence of integers. While it may be possible to generate an index value similar to that of figure 3.14 using C* operations on the rank-2 shape, it would not be nearly as succinct and efficient as the shape alias plus enumerate method used in the example.

It is this need to be able to represent blocks of data as subgrids in differently-ranked shapes that mandates the support of arbitrary partitioning of axes described in section 3.1, so that the code in figure 3.12 will work regardless of the number of rows and columns in the image, the distribution of the two-dimensional image, and number of workers in the cluster. If the user, for independent reasons, distributed `Image` only along axis 0, and the system required a near-equal partitioning along the axis (where each node had at most one more row than any other node), the difference of one row would translate to a difference of `dimof(Image,1)` in number of positions, and the near-equal partitioning would not be satisfied in the one-dimensional alias.

3.2.4 Data Access Patterns

The fact that shape dimensions are unknown at compile time means that the virtual processor loops which perform computations at all elements in the shape walk through data in their linear form, disregarding rank and extents. It is not possible for the compiler to emit a nested loop structure to walk a parallel value, unless one is willing to provide at compile time a hard upper-bound on the rank of shapes on which the generated code will operate. An algorithm outlined in section 6.1 shows that it is possible to emulate arbitrary for-loops using a while loop, but it is unnecessary in the cases examined so far to do so, since a linear walk provides a perfectly adequate access pattern (though this does require us to use a more complex `pcoord` computation than simply referencing an index variable).

However, there are a variety of auxiliary C* functions which operate on one user-

data

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

dx0

	0	1	2	3
0	60	64	68	72
1	?	?	?	?
2	?	?	?	?
3	?	?	?	?

dx1

	0	1	2	3
0	?	?	6	?
1	?	?	46	?
2	?	?	86	?
3	?	?	126	?

Figure 3.16: Auxiliary Reduce Function Example

specified axis of a shape at a time. For example, the library `reduce` function[9] takes as parameters a shape, an axis number, a destination index, and an operator. It then separates the shape into groups of elements where each group has the same global index except within the provided axis, applies the operator to the elements in each group, and stores the answer in the position that is at the given index along the axis. These groups are known as *scansets*, and, in a 2d shape, correspond to rows or columns. For example, figure 3.16 shows the application of the add operation to a 2 dimensional shape, with one result being along axis 0 (scansets are columns) and stored in index 0, and the other along axis 1 (scansets are rows) stored in index 2.

The simplest way to implement this function is to use nested for-loops, one loop for each axis and ordering the loops so that the innermost one iterates over the axis along which the reduction is performed; e.g., for the axis 0 reduction in figure 3.16 the effective code would be something like:

```
for (c = 0; c < 4; c++) {
  sum = 0;
  for (r = 0; r < 4; r++) {
    sum += data [r][c];
  }
  dx0 [0][c] = sum;
}
```

Although the reduce function is a library routine so cannot have hard-coded nested loops, the multi-dimensional for-loop emulation method of chapter 6 can be modified to provide this functionality. The primary benefits of this method are that it is conceptually simple, and it requires only one scalar temporary to hold the accumulated results along the scanset that is currently being walked.

The primary disadvantage of the method is that it has extremely poor memory access patterns. Rather than perform one pass over the local data set, it will perform multiple passes, each one accessing a slightly different area of memory. Advances in computer hardware mean that access time increases by several orders of magnitude as the data that are required

9. This routine is one of the `cscomm` auxiliary routines provided in the Thinking Machines Corporation implementation of C*, and should distinguished from the core language reduce function and the pC* internal `reduce` collective communications routine described in section 4.3.2.

Dimensions	Axis	AXIAL	LINEAR
1000000	0	0.1501	0.2076
100×10000	0	0.5017	0.1949
1000×1000	0	0.6509	0.1720
10000×100	0	0.7677	0.1726
100×10000	1	0.1500	0.2077
1000×1000	1	0.1507	0.2079
10000×100	1	0.1614	0.2115
$100 \times 100 \times 100$	0	0.5076	0.1936
$100 \times 100 \times 100$	1	0.1935	0.1769
$100 \times 100 \times 100$	2	0.1631	0.2115

Table 3.3: Axial versus Linear Walks of Multidimensional Data. Time in seconds over one million positions.

live further away from the processor along the spectrum of on-processor cache, external cache, main memory, and paged disk memory (Hennessy & Patterson, 1990). The large-stride access means that the block of memory holding a desired element must be loaded into the cache, incurring some cost, but we then refer only to the one element in that block, losing the chance to amortize the load cost by operating on all elements in order. In the case of large images, the amount of data is such that while walking along a column performing a reduction the previously-loaded blocks must be freed to make room for more data, so the load cost is incurred again on the next column.

To quantify the overhead involved, we implemented the reduce function in sequential C, and tested the performance on a variety of one million element shapes ranging from 1 to 3 dimensions on each axis. The experimental platform was a Sun Sparc 20, using a 50 MHz SuperSPARC chip with a 16K 4-way set associative D-cache on-chip, a 1MB second level cache, and 512MB of memory. General algorithms were coded to handle arbitrarily-dimensioned shapes using one of two methods: AXIAL emulates the above nested for-loop method operating on each scanset in sequence, while LINEAR walks the data in their locally stored order, and uses a variant of the step/wrap `pcoord` method to determine which scanset it is examining. The LINEAR algorithm therefore requires an array which maintains the partial sums of many scansets simultaneously. Results of the experiment are shown in table 3.3. Walking axis $k-1$ of a rank k shape using the LINEAR algorithm induces a roughly 15% performance loss (relative to other axes) due to the same effect seen in the step/wrap `pcoord` calculation on this axis: specifically, overhead from repeatedly executing a loop for one iteration, because a special case could not be anticipated. The effect of bad memory access can be seen most clearly in the lower axes of multidimensional shapes. The degree of interference is proportional to the number of times a different region of memory is accessed: hence using AXIAL on the 100×10000 shape on axis 0 the inner loop iterates only 100 times before it finishes and data it has already loaded is accessed for the next scanset. But when the shape is 10000×100, there are 10000 accesses before re-examining a previously-loaded cache line, and the likelihood of the data surviving in the cache is correspondingly less, increasing the average access cost. The inverse interference pattern can be seen in the performance of LINEAR on the same data: for the first case the summary vector is 10000 elements long, yielding a much larger cache footprint than the 100 element summary vector for the second case.

Assuming the likelihood of walking each axis is equal, the AXIAL method on the square rank-2 shape takes on average 0.4 seconds, compared with 0.19 seconds for the LINEAR method. Thus, by carefully coding all algorithms to walk data in a cache-sensitive manner we can obtain very good speedups. In addition, mandating this access pattern (when we must touch every element in a shape) allows additional optimizations such as the context encoding described next.

3.3 Implementation of Context

Context allows the C* programmer to specify that operations should not be performed on certain elements in the shape. Essentially this associates a boolean value with each position, indicating whether or not the position is active for operations. As all operations not within the direct scope of an `everywhere` block are performed under context, it is important to make the overhead of context checking as small as possible.

3.3.1 Representation of Context

The natural first approach to context is to use a parallel boolean value. Since bit-wise operations on modern RISC processors usually require multiple instructions, a sensible alternative is to use one byte per position, with 0 indicating an inactive and non-zero an active position. Thus, inside a VP loop, a test can be made to determine whether the operation should be performed for that virtual processor: e.g.

```
/* OMITTED: initialize other pvar pointers and strides */
PCS__ctx = (char *) PCS__current->context;
for (PCS__vpi = 0; PCS__vpi < PCS__vplimit; ++PCS__vpi) {
  if (*PCS__ctx) {
    /* OMITTED: perform operation here */
  }
  /* OMITTED: add stride to pvar pointers */
  ++PCS__ctx;
}
```

While simple, this method, which we will call a *charmap* encoding, is suboptimal for a variety of reasons that become clear when the common forms of context are considered. Contexts generally fall into one of the following categories:

- Everywhere—Operations that have no inactive positions. In this case, we pay the price for storing a charmap which is all ones, and for checking it at each position. Furthermore, although the compiler may be able to recognize that code appears within the boundary of an everywhere context and thus generate a VP loop that omits the unnecessary check, library routines will be unable to take advantage of this information unless everywhere contexts are marked internally in some form.

- Boundary elimination—An internal regular region of the shape is active, but there is a boundary region for each axis which masks out areas which would induce out-of-bounds grid accesses or destroy values that are to be algorithmically invariant. These contexts generally alternate long runs of active positions with short runs of inactive positions.
- Random—There is no easily-recognizable structural regularity to the context. Often a high percentage of the shape is inactive, as with the high-elevation context in the example on page 41.
- Tiled—Elements are alternately on and off: for example, as would be used in a red-black or odd-even simultaneous over-relaxation algorithm (Press, Flannery, Teukoloky, & Vetterling, 1984).

We counted the uses of context restrictions in the source for benchmarks described in chapter 7, and found fifteen everywhere (in part because of the performance benefits of doing this when entering a new function, to permit extra compile-time optimizations), thirteen boundary elimination, and nine random or non-structured contexts. Tiled contexts can appear in some numerical analysis algorithms but are virtually unknown in image processing applications. The other forms are fairly common, and have a noticeable feature that they have long runs of consecutive elements with the same context type (active or inactive). Given that compiler-generated VP loops and library routines both access data in internal, linear order, this leads one to consider whether a run-length encoding of context might be more efficient, both in terms of space required to store the context, and in time by skipping over inactive regions in one step and putting operations over active regions inside a tight loop which does not need to check context.

A run-length encoding (RLE) of context can be implemented by using a signed integral type to encode the run, with positive values indicating an inactive sequence and negative values an active sequence (or vice-versa). The RLE context map is represented by a pointer to a sequence of integers which encode runs; the space required by the map depends on the integer data-type used to encode runs, and the variability of the context. By combining adjacent runs of the same type in the function which reads the context map, we can get the full length of each active and inactive sequence, allowing a simple conditional to distinguish an inactive region that should be skipped immediately from an active region that does not require context checking. An `everywhere` context can be easily represented by a null pointer, which the routines interpret as an active sequence of `num_local` positions. This allows both an optimal space representation and communication of the `everywhere` context into library routines and user functions where the compiler isn't able to determine the execution context at compile time, at the cost of only a single calculation at VP loop entry.

The next question is what size value should be used as the base type. If we use 8 bit chars, which can represent from -128 to 127, we are guaranteed that in the worst case

Program	# Everywhere	# Where	char RLE	short RLE	int RLE
amp	2	4	0.01302	0.01270	n/a
col	0	384	0.22298	0.44370	n/a
dens	1222	1827	0.00896	0.00286	n/a
fft	1	65	0.12753	0.24615	n/a
hist	100	0	n/a	n/a	n/a
hough	3	0	n/a	n/a	n/a
hyp	111	200	0.20903	0.41674	n/a
jac	1	1	0.00788	0.01576	n/a
mat	111	43	0.30528	0.60655	n/a
obj	2	2	0.21810	0.43620	n/a
det-arag	25	41	0.01655	0.01914	0.03827
det-famp	18	35	0.01515	0.01651	0.03298
cta-arag	41	736	0.09490	0.18156	0.36313
cta-famp	47	582	0.10705	0.20536	0.41072

Table 3.4: Context Encoding Frequency and Space Summary. Number of dynamic context formations of each type, and fraction of space required by RLE relative to the charmap encoding.

(a tiling where each run is 1 element long) the RLE encoding will take no more space than the charmap encoding; while if the entire local subgrid is inactive, we must use $\lceil \texttt{num\_local}/127 \rceil$ bytes to store the context. A 32 bit integer encoding with a representation magnitude of at least 2^{31} would eliminate the second problem by encoding the entire run length in one word, at a potential cost of quadrupling the space requirements for a tiled context. Using a 16 bit short word is an intermediate alternative.

We instrumented a set of 10 test programs (from (Turner, 1994)) and four full image processing programs to determine the effect of using run-length encoded contexts with various data types. The results are shown in table 3.4, and give the dynamic numbers of contexts—separated into `everywhere` and non-everywhere—generated during one run on a single representative input for each program on a single-node cluster. The data-sets were small to medium-sized problems for each program, ranging from a 256×256 sub-image for hist, a 10 band 128×128 sub-image for hyp, up to 2048×1024 for det. The context maps were examined, and the space requirements to represent the non-everywhere contexts using run-length encodings in each of the basic types—char (1 byte), short (2 bytes), and int (4 bytes)—normalized to the charmap encoding (one byte per element), are presented. It is clear that an RLE encoding with `char` base types is sufficient: although two of the fourteen programs (amp and dens) got a small space improvement from using `short` base types, on average using shorts increased memory requirements by a factor of 1.93, very nearly the worst-case increase of $2\times$. Therefore, for these types of programs, there is no benefit in using more than 8 bits for the run-length encoding type: enough of the runs are short enough that the increased representation range is unnecessary. In addition, the relative space requirements show that RLE encodings tend to consume either around 1% or 10–20% of the space of the

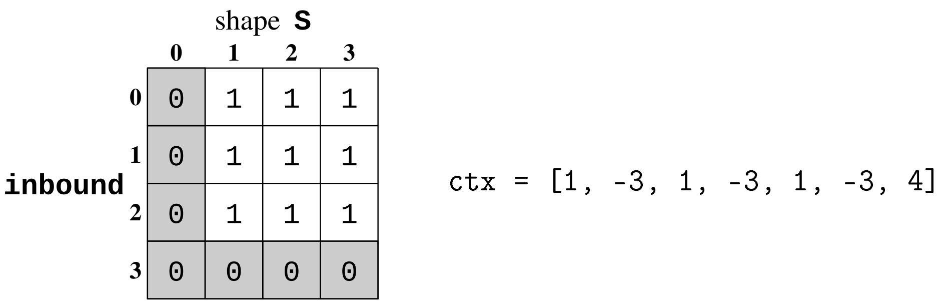


Figure 3.17: Encoding of Boundary Context

full charmap encoding on these programs: this does not count the savings from representing `everywhere` contexts as null pointers. (To help in understanding the effectiveness of this on overall memory usage, other instrumentation on the image processing programs indicated that the charmap context encoding consumed from 2% to 7% of the peak dynamic memory used by the program.)

As an example of context encoding, consider the boundary restriction required to avoid out-of-bounds access while executing the following code:

```
shape [4][4] S;
...
with (S) where (inbounds) {
  iv2 = [.+1][.-1]iv;
}
```

The source variable is shown in figure 3.17, along with the run-length encoding assuming all elements appear on one node. The corresponding C code which performs the assignment under context is shown in figure 3.18. This is the general method for implementing contexted VP loops with run-length encoded context. `PCS__ctx_nextseq` is a macro used to determine the combined length of a context run. It detects an `everywhere` context by noticing a null context pointer, and handles it implicitly. If the context is not `everywhere`, it will combine adjacent runs of the same context type (active/inactive) yielding the total length of the run. It updates the `PCS__ctxcnt` variable to give the length and type of the next run. It requires `PCS__vpi` and `PCS__vplimit` to ensure it does not go beyond the ends of the context encoding, since the length of the encoding is not stored elsewhere.

3.3.2 Building Context

Once it is decided to use a run-length encoded context representation, methods for building the run encoding are fairly clear, though efficient implementation can be nontrivial. We do not know the space required to hold the encoding until it is complete. Therefore, the routines that allocate shapes ensure that there is a memory buffer called the *context build arena* which is as large in bytes as the maximum number of local positions in any shape. The choice of `chars` as base elements, along with care in the generation method to ensure

```
/* OMITTED: build the context */
/* OMITTED: perform grid read into PCS__gtmp */
/* OMITTED: initialize pvar pointers and strides */
PCS__ctx = PCS__current->context;
while (PCS__vpi < PCS__vplimit) {
  PCS__ctx_nextseq (PCS__ctxcnt, PCS__ctx, PCS__vpi, PCS__vplimit);
  if (0 > PCS__ctxcnt) {
    do {
      *(int *)PCS__iv2p = * (int *)PCS__gtmpp;
      ++PCS__vpi;
      PCS__iv2p += PCS__iv2s;
      PCS__gtmpp += PCS__gtmps;
    } while (0 > ++PCS__ctxcnt);
  } else {
    PCS__vpi += PCS__ctxcnt;
    PCS__iv2p += PCS__ctxcnt * PCS__iv2s;
    PCS__gtmpp += PCS__ctxcnt * PCS__gtmps;
  }
}
```

Figure 3.18: Code for RLE-Contextualized Parallel-Value Assignment

no 0-length runs are stored, ensures that this will hold any context which might be required.

Context builds proceed in an inverse of the way the context is used: we determine the length and type of a context sequence, then store the RLE encoding for it. The simplest case is building an initial context restriction: a `where` construct which appears directly in scope of an `everywhere` block. The code in figure 3.19 shows the build process, where `expr` represents some parallel boolean expression. While the control flow may seem somewhat baroque, it serves a purpose. Notice that we evaluate `expr` once for each position, regardless of whether at the time we correctly anticipated the value of the expression: if we were wrong, we store the now completed sequence, change the sense of the current sequence and reset the count, then jump back into the loop and continue. This also allows us to avoid duplicating the code for incrementing the pointers which walk parallel values referred to in `expr`. We take care to ensure that only non-zero values are stored in the context, so we do not exceed the space bounds of the arena. At the end of the build code, we allocate a block of `ctxp-context_arena` bytes, copy the new encoding into it, save (the pointer to) the previous context in a variable in the block, and store the new context map in the shape. If, after the `where` expression has executed, there is an `else` expression which requires the inverse context, we need only walk the map changing the sign on the saved run lengths.

Building a cumulative context is substantially more tricky, since we must preserve the inactive sequences from the parent context, yet add new ones where the new restriction is more stringent. Essentially the build loop is lifted into a VP loop contexted on the parent context: inactive sequences in the parent are immediately stored, while active sequences proceed to execute the test code and store active or inactive sequences according to the

```
vpi = 0;
vplimit = PCS__current->pernode[mynode].num_local;
cnt = 0;
testval = 0;
ctxp = context_arena;
/* OMITTED: initialize pointers and strides for parallel variables
 * in test expr */
while ((vpi < limit) && (testval == !!(expr))) {
ctx_lbl1:
  ++cnt;
  ++vpi;
  /* OMITTED: increment pointers for parallel variables in expr */
}
while (RLE_Limit < cnt) {
  *ctxp++ = testval ? -RLE_Limit : RLE_limit;
  cnt -= RLE_Limit;
}
if (0 < cnt) {
  *ctxp++ = testval ? -cnt : cnt;
}
if (vpi < vplimit) {
  testval = !testval;
  cnt = 0;
  goto ctx_lbl1;
}
```

Figure 3.19: Code for RLE Context Formation

boolean expression results. Care is taken to combine adjacent runs to decrease space: if a parent inactive sequence is $k * \texttt{RLE\_Limit} + i$ long, the extra i will be combined into any initial inactive sequence resulting from false `expr` values in the new context. The required contortions are not sufficiently interesting to reproduce here. There are similar difficulties in producing a cumulative `else` context, since we need to take the inverse of the child encoding while preserving inactive sequences in the parent: this becomes a sort of merge operation, walking child and parent simultaneously and switching sequences which are active in the parent, again merging adjacent runs.

3.3.3 Additional Context Optimizations

In addition to a run-length encoding, several steps can be taken to decrease the cost associated with context. In many cases, it is not necessary for the context to be stored at all: if the body of the `where` (and/or `else`) construct is all parallel code, then the test can be moved into the virtual processor loop and executed for each position. Similarly, any initial portion of the `where` body which would appear inside a VP loop can be lifted into the context build loop, at the point immediately following the `ctx_lbl1` label in the build loop in figure 3.19, avoiding the need for an extra loop construct immediately after the context build.

Type	Operation	charmap	RLE
`everywhere`	build	0.10964	0.00002
	body	0.24631	0.20538
boundary	init build	0.18265	0.16101
	cumulative build	0.20747	0.15543
	body	0.24429	0.20094
random	init build	0.18612	0.20545
	cumulative build	0.17261	0.08345
	body	0.16968	0.08376
tiled	init build	0.10592	0.54829
	cumulative build	0.18654	0.81277
	body	0.25197	0.53542

Table 3.5: Context Build/Reference Timings. Seconds for build/reference in rank-1 shape with 2^{20} positions.

These features have been implemented in pC*, with some amount of additional complexity to the context building code described in the previous section.

A significantly more powerful context optimization is in handling of the boundary elimination case. If the `where` expression consists of a conjunction of comparisons between `pcoord` and a scalar integral expression, the resulting context can be generated directly using techniques similar to those of grid communication in chapter 6. In essence, we emulate a multidimensional loop over the local subgrid, with lower and upper bounds corresponding to the region of the subgrid owned by the node, but corrected for range restrictions imposed by the `where` expression. When an index of the iteration space wraps, we have reached an out-of-bounds area, and store into the context an inactive sequence whose length depends on how many axes wrapped and the bounds of each of them. A full description of the technique is reserved for chapter 6, since it is based on the same techniques as the method for skipping or detecting out-of-bounds grid axes described therein. Since evaluating `pcoord` at each element can be expensive even with the step/wrap implementation of section 3.2.1.1, this technique results in a significant speedup.

3.3.4 Evaluation of Context Optimizations

We have already seen from table 3.4 that run-length encoding can result in a 90%–99% decrease in space requirements. However, it should be clear that the build loop is more complex than that for a charmap encoding, and the loop nesting required for contexted VP loops is also more expensive than a simple boolean test. We should therefore examine the effect of a run-length encoding on execution time.

Table 3.5 contains information about context build and reference for RLE and charmap implementations of each of the context classes mentioned earlier. In all cases, the experimental body consists of a call to a function which assigns from one parallel variable to another. Therefore, within the function the execution context is unknown, and no inlining of bodies into context build loops was done, nor was context check code eliminated due to a syntactically enclosing `everywhere` block. Except for `everywhere` the contexts were built from a pre-computed boolean variable which reflected the type of mask—in essence, a charmap encoding itself, although neither implementation took direct advantage of this in forming the context. Timings are in seconds, and are the median of five runs on a 50MHz Supersparc. Operations were performed on a 1024×1024 shape. Builds for initial contexts recognized that the parent context was `everywhere`, and avoided the merge step; cumulative contexts built the initial context, then timed the cumulative build using the same mask again.

- Everywhere—Build time for charmap requires allocating a block of memory and initializing it to all 1s. For RLE, it requires assigning a null pointer, a constant-time operation regardless of shape size. The improvement in RLE body time is due to the avoidance of checking the charmap flag for each position; if the test had permitted the body to be syntactically enclosed in an `everywhere` context, the compiler would have noted this and the charmap time would be nearly identical to the RLE time.

- Boundary—The leftmost and bottommost 5 columns and rows were disabled, leaving 99.03% of the shape active. Notice that the initial build times for both methods are approximately the same, while the cumulative build was faster for RLE, due mostly to avoiding the context check for the active runs, and possibly aided by better cache behavior with the smaller context encoding. Execution of the body is 19% faster with the RLE encoding, because context need not be examined for long runs.

- Random—5% of the internal elements, chosen at random, were made active, with the rest inactive. The resulting map is not an accurate reflection of the type of "random" contexts expected in image processing, because images would tend to group active areas into longer active and inactive runs, providing better amortization during builds and loops. The initial build is slightly more expensive for the RLE encoding, but the cumulative one is twice as fast, mostly because it can skip the 95% inactive positions where the charmap encoding must at least look at them. Using the resulting RLE context is also twice as fast as using the charmap, for the same reason.

- Tiled—Odd numbered elements were active, while even were inactive. This is the worst case for the RLE context: it consumes as much space as the charmap encoding, and the overhead of storing and extracting the unit sequences, which is not amortized over long runs, causes a four-fold slowdown during builds, and just over two-fold slowdown during VP loops.

While the results in table 3.5 reflect a single simplistic test, experience with test and production programs during development of the encoding tended to support a 10%–25% improvement in execution time using the RLE contexts. This is mostly because the worst case of tiled context does not occur in the image processing applications we currently use. In no case did the RLE context result in a significant slowdown, and in all cases it saved more than half the space required by charmap encodings.

The times in table 3.5 indicate the cost of building a boundary context when the source expression is a boolean variable and each position is tested. In many cases, especially for the boundary context, the expression contains calls to `pcoord`. We describe in section 6.1 a method to build contexts more quickly in this case, i.e. without evaluating `pcoord` for each element of the shape. For the experimental results in table 3.5, the time to build the boundary context evaluating `pcoord` at each position (using the step/wrap method) was 0.34sec, less than twice the time required to examine the pre-computed boolean value. When the aforementioned optimization which takes advantage of grid restrictions to build boundary contexts is used, the build time drops to 0.0029 seconds, over 100× faster: rather than performing a test on each of one million elements, it need merely execute a loop body 1024 times and store a count on each iteration. Tests on other programs indicated that the time required to build a boundary-safe context without the optimized build was approximately half the time required to perform the grid communication it protected; with the optimized build, the context formation time becomes insignificant. The optimized build method was never applied to the charmap case; while it would undoubtedly provide a significant speedup there as well, one would expect to lose an order of magnitude due to cache effects required to store a value at each of one million bytes, similar to the cost difference observed for building an `everywhere` context in the charmap style.

It is clear that, if the problem domains to be addressed by a system include frequent use of tiled contexts, an RLE encoding may not be appropriate: in fact, given the RLE encoding, it is faster in pC* to perform simultaneous overrelaxation operations at all positions in the shape, then generate the value for the next step using a tiled context which can be wholly inlined, avoiding the costly build/walk steps at the price of ignoring half of the computed results. Alternatively, one can simply split the problem data into two separate pvars, one each holding red or black data, and perform the computation by alternating between them without contextualization. Since all functions which examine data must know the format in which context is represented, and an RLE representation is significantly different from a charmap encoding, it is impractical to support both in the same system.

3.4 Conclusions and Related Work

We've considered three major issues in language implementation in this chapter. Much work on distributed computation presumes that data distribution is a critical component to system performance (Tseng, 1993). While we do not wish to argue that this is untrue, it is clear that data placement is not the whole story: we must also be able to convert between global and local addresses, to know what values must be transferred between processors. Even the fairly simple block-based distribution scheme discussed in this chapter can, if not carefully optimized, require a significant proportion of execution or communication time just to perform the conversions. Less regular schemes such as those described in (Crandall & Quinn, 1993; Socha, 1991), which are designed to decrease the high-level communication requirements by partitioning data with less regard to topological neighborhood or orthogonal boundaries, would intuitively require more effort for the conversion phases; the descriptions of these placement methods do not address this issue. The appropriateness of more complex placements for particular algorithms, such as block-cyclic for linear algebra routines, should be tempered by the effect on the rest of the system of adding this support. When all communication patterns and address resolution can be discerned at compile time, this is not such a large issue, but it will affect general case implementations that are relied on when compiler analysis fails to detect an optimizable case.

We have also examined the value of interpreting the locally-owned data as linear sequences, regardless of the layout imposed on them by the user. This is done in the C code generated from user C* code, in part because the necessary information (rank and extents) for generating nested loops for user-view access may not be available to the compiler (though techniques described in chapter 6 would permit a simulation of nested loop if that were useful). However, linear access in library routines is primarily desirable in order to improve cache behavior. Fortran source usually presents sufficient information for the compiler to emit a nested-loop access scheme, but for code fragments such as DOALL loops where the semantics does not require preserving the user's access pattern there may be a significant benefit to re-ordering or linearizing the loops so data are accessed in one-pass sequences.

Issues of data access and address computation in the context of a sequential translation from APL to C are considered in (Budd, 1988). Here the interest is more in decreasing memory usage, and a choice is made to use a non-contiguous access pattern for complex primitive operations that would otherwise require space for temporary intermediate values. The work also addresses the need for fast address calculation, though in a uni-processor implementation and therefore not to the generality described here.

Finally, we have presented a method of encoding context which requires much less space than the "natural" charmap encoding and reduces overhead in loops by making the cost of context checking proportional to the number of context sequences rather than the number of elements in the shape. Although this encoding is valuable for the context types common to image processing applications, it is clear that algorithms which use tiled contexts, as in red-black simultaneous overrelaxation systems, can behave poorly. Implementers should consider the target applications of their systems and choose the appropriate encoding accordingly. Unlike the previous two issues which can be applied to languages such as Fortran, the C* notion of context is more general and requires communicating the activity information to operations distant from the context formation, while Fortran 90 context is a purely local phenomenon which does not reach into called functions, and it is not clear that storing the context is required in that case.

CHAPTER 4

BASIC COMMUNICATION PRINCIPLES

Are you sitting comfortably? Then we'll begin.

— Julia Lang (introduction to BBC Radio programme *Listen with Mother*)

In this chapter we examine in detail fundamental assumptions about interconnection networks in the context of a highly portable yet high performance distributed language. The three-level framework of pC's communications hierarchy is described. A low-level interface permits target-specific implementation of a limited number of functions, and delivers a common quality-of-service when its requirements are met. At the highest level are language-specific communications operations, such as grid and general communications patterns and more complex computations such as sorting and prefix scans. Mediating these levels is a target-independent set of routines which use the low-level interface to implement communication of arbitrary amounts of data, as well as basic group communications such as one-to-all broadcast and all-to-all reductions. We describe how a message handling facility built into the mid-level routines is required for correctness, but simultaneously reduces memory usage and will permit eventual implementation of latency-hiding communications optimizations, given sufficient compiler support. We close with a case study which implements broadcast and reduction operations with a variety of algorithms on both point-to-point reliable Unix TCP sockets and multicast unreliable Unix UDP sockets, coming to the conclusion that the benefits of a broadcast/multicast implementation, given their frequency of use in our benchmark suite, are negligible relative to the system complexity that is introduced by imposing our reliability requirements on them in a portable manner.*

Distribution of data amongst nodes in a distributed multiprocessor system is a major factor in the performance of the system, but regardless of the distribution chosen data will eventually need to be moved between processors. This implies that the communications subsystem used is of similarly high importance to total system performance. However, the path to choosing an appropriate implementation framework is difficult because network issues—especially in a system that attempts to be portable—are more complex than data distribution, a matter that is purely internal to the system. In this chapter we will examine the issues that led us to the communications system that is used in pC*, and describe the primary features and design considerations of the system. We conclude with an extended look at the relative performance of two network-level message facilities to show how reliability, features, and performance must be balanced to meet the needs of the pC* system.

Throughout this chapter, by "network" we mean any mechanism for communicating between processes. Most often this takes the form of connecting machines through a physical path such as Ethernet, FDDI, or ATM, but it also includes Unix domain sockets, System V messages, shared memory, a shared file system, or anything else upon which the functionality demanded in section 4.3.1 can be imposed. The "network interface" is the set of system-provided functions or capabilities that are used to interact with the network.

4.1 Portability versus Performance

The issue of connecting computers together to cooperate in tasks through some form of internetwork is a major topic of investigation in computer science. There are virtually limitless ways to approach the issue for a particular application, depending on the level at which one cares to address the problem. Sufficient research issues exist in this component alone that without sacrifices and compromises of some sort the primary goal of a portable and reasonably efficient implementation framework for data-parallel languages would be unreachable. Here we examine the alternatives across the spectrum from highly portable communications packages which may be unable to take full advantage of particular platforms, to highly optimized implementations which use the features of a particular host configuration to provide excellent performance while forgoing any hope of portability. While there are research efforts to create systems that collapse the spectrum (Message Passing Interface Forum, 1994; Mitra, Payne, Shuler, van de Geijn, & Watts, 1995), those efforts have not yet succeeded fully, preventing us from relying on them.

4.1.1 General Purpose Communication Libraries

In recent years, the cost of computer hardware has decreased, and scientists and other programmers wish to connect a variety of small machines to solve problems that formerly required large, expensive, and dedicated hardware. Issues of ease-of-programming and compatibility across these systems have led to the development of a variety of libraries which provide functions to aid in distributed computing across homogeneous or heterogeneous clusters of computers linked with various networks. These libraries provide routines for joining computations, passing data between computation processes, and performing high-level collective communications such as global reductions without burdening the applications programmer with the need to implement these herself. Perhaps the most well known of these is currently the Portable Virtual Machine (PVM) of ORNL (Geist, Beguelin, Dongarra, Jiang, Manchek, & Sunderam, 1994), though industry preference now seems to be turning towards the Message-Passing Interface (MPI) (Message Passing Interface Forum, 1994) which combines features of PVM with those from similar libraries in an attempt to provide a standard library that will be supported by all vendors of high performance computing devices.

The primary goal of these libraries is to provide access to a set of common functions while ensuring portability of the applications which use them. PVM has been adopted by several vendors who have provided optimized implementations which take advantage of the particular features of their hardware platforms. Platforms which do not have specialized implementations are still able to use the routines through standard hardware (e.g., Ethernet) using common networking interfaces (e.g., Unix sockets). The PVM source comes with configuration files to support fifty different platforms. Since portability is a major goal of the pC* system, it is worth considering the use of a general library such as PVM or MPI as the communications infrastructure of the system.

Though the benefits of using such libraries seem clear when coding applications using languages which were not originally intended to provide support for multiprocessor computing, such as C and Fortran, general communications libraries appear to be ill-suited as a component of a language system which already has parallel constructs. Though there are routines provided to perform some high-level operations, such as global reductions, these may not have the semantics required by the parallel language, such as following the C* notion of context. There are other C* routines, such as the segmented scan operations, that are not present and would need to be implemented using more primitive operations from the library. Furthermore, the library necessarily constrains data layout and internal representations to match its own requirements. If all these issues are addressed in the context of the communication requirements of the language, the only components of the general library that are really useful are simple point-to-point message exchange, and perhaps some form of broadcast.

An additional problem is the performance of the general system. Although vendors may provide an optimized library for high-cost hardware such as the Cray T3D (Oed, 1993), they are less likely to do so for lower cost systems, such as networked workstations or PCs. The effort required of the library implementors to support the whole library takes away from the opportunities to make the few routines required by the pC* system run fast. For example, we compared the execution time of PVM with that of a plain TCP socket implementation on two 4 processor Sun SS20s connected by Ethernet, in a C* program which "transposed" (reversed) a distributed 1-dimensional parallel variable of 2^{16} elements ten times: a test which involved almost nothing but constant point-to-point communication. Even though the PVM system was informed that it was on a homogeneous network and need not encode its parameters (`PvmDataRaw`), it was 35% slower (4.10 seconds versus 3.03 seconds) on four processors on the same machine, and 130% slower (10.8 versus 4.67 seconds) on eight processors across both machines, than the plain socket implementation. Slow-downs of this magnitude are a high price to pay for portability, especially when already buying unneeded functionality.

These issues led us to avoid attempting to use a general purpose communication system as the basic framework for the communication needs of the pC* runtime libraries, and network and application-specific concerns outlined below drove the system design. For completeness and comparison purposes, though, PVM support has been added to the pC* system, and hence the communications portion of pC*—which is the most system-dependent component—may be ported quickly to any system which implements PVM.

4.1.2 Direct Network Control

At the other end of the spectrum are communications infrastructures which are based on the characteristics of particular networks, and permeate the entire communications system with that knowledge. This could include assumptions about non-standard interconnects such as hyper-cube routing networks, which are more common on high speed massively parallel systems such as the TMC CM5 or Intel Paragon. Performance on these interconnects can be improved by ensuring that operations are performed in a way to reduce network congestion by delaying or re-routing message transmissions which could induce collisions, at a potentially significant cost in code complexity and portability in our case.

Another possibility is to assume that the host architecture provides some means of direct access to the network device, a feature which is unusual in standard operating systems because of concerns for security of network data, but can be countenanced when considering an isolated compute cluster dedicated to a single task (Turner, 1994). Without such direct control, any program running at user level suffers due to the overheads involved in crossing protection domains to invoke code which is permitted to drive the network controller, and in copying data between domains and processes (Druschel, 1994; Brustoloni & Bershad, 1993). A significant performance improvement could be induced if the C* system were able to have data transmitted directly between the network adaptor and the memory locations where they are stored in the C* program's memory space. However, these features are generally available only in research operating systems, or to those who have access to the vendor source (Chang, Flower, Forecast, Gray, Hawe, Nadkarni, Ramakrishnan, Shikarpur, & Wilde, 1994).

If one is constrained to write code which can be executed by a normal user on a normal operating system, in accordance with the pC* design goals, there are still restrictions that could be made to improve performance. For example, we may choose to assume that we will run on a Unix-based computer, and can use standard Unix system calls to access network functions through the streams or socket libraries. In this case, we will probably be able to use interrupt driven communications, where the kernel of the operating system will detect the arrival of a message at a network interface, and asynchronously inform the library system of its presence. The system can then immediately read the message and deal with it appropriately, relieving the kernel of the burden of buffering the message until the C* system gets around to looking for it. However, even assumptions such as this would cause portability and usability problems for the pC* system. Since the entire system is operating at the same protection/user level, installing a signal handler removes a useful resource from the programmer, who would no longer be able to use the same interrupt-driven I/O in her own programs without interfering with basic system functionality. It would also be necessary to ensure that the invocation of a message handler without warning could not corrupt any data structures in the program, either in the library or in the user's code.

Finally, even such relatively weak assumptions turn out to affect portability. On shared-memory multiprocessors such as the Sequent Symmetry, or even distributed multiprocessors such as the Intel Paragon, the preferred method of communicating between processes may not involve Unix system calls, and may not have a notion of interrupt-driven communication. By designing the system around those assumptions we would have limited its portability to such machines. During the first year of development of the pC* system, we examined the requirements necessary to port the system to machines ranging from the Intel Paragon to multiprocessor SGIs to Linux-based PCs. The assumptions we ended up using allowed us to eventually perform these ports with limited effort.

4.1.3 Application Specific Libraries

The "middle ground" approach to networking for pC* is to design a library of communications routines which are specialized to the needs of the C* language, but not to any particular assumptions about the connection mechanism. By designing a hierarchy to isolate the components of the library which are network specific, we get the best—and worst—of both worlds: the bulk of the library is platform-independent, and only the lowest level routines need to be written when porting the system to new hardware, while we are able to take advantage of our knowledge of communication behavior within the system and the general features of the expected target architectures to avoid unnecessary overhead. In return, we pay the penalty of not taking full advantage of certain features such as direct adaptor control when the low-level functions where we have access to these features are too far from where they would do the most good.

It is safe to say that any inter-process(or) interface will have certain limitations on the amount of data it can handle at any time, or the quality of service that it guarantees. However, given the complexity of some of the algorithms that must be generated, such as the grid operations described in chapter 6 or the parallel prefix scan operations not explicitly described in this dissertation, it is important to limit the effect of these constraints on the higher level functions. We can now go on to examine the restrictions we were willing to make on the system, and how they affected the design of the communications hierarchy within the pC* runtime system.

4.2 Network Assumptions

We must pause to review the philosophy under which pC* was designed, and how this affects the assumptions we are free to make about the execution network environment. First, the most important aspect of the system is that it must perform correctly, and operate on its intended application—analysis of large images—without crippling limitations. This means, for example, that limitations on the amount of data that can be buffered by a network interface must not be allowed to interfere with system behavior: we cannot permit deadlock if a kernel buffer would overflow and a system call block just because a user chose to invoke an operation which sent a 32MB chunk across the network. Nearly as important is the requirement that it be portable, so that we would not find ourselves bound permanently to a particular hardware platform again. We can make only the most general assumptions about the implementation platform. Performance is the third most important goal. These issues led us to design a three-level communications hierarchy:

- High-level functions include the C* grid and general communication routines, as well as complex library routines such as `scan` and `spread`.
- Mid-level functions are an abstraction of the unconstrained network interface, including writing messages to other nodes, broadcasting, and performing simple reductions.
- Low-level functions implement a constrained interface to a particular communications network, taking responsibility for providing any service requirements that are assumed in the higher level routines but are not provided by the network.

Assumptions about network characteristics can reach throughout this hierarchy or be isolated in one level, depending on their effect on the system as a whole.

Let us first consider the issue of message size. Almost every network fabric will have some limitation on the amount of data it will operate on as a single entity. This is the network's *maximum transfer unit*, or MTU. It is very likely that the MTU will be small compared with the amount of data being transferred: for Ethernet, it is 1500 bytes, while for ATM it may be as small as 56 bytes (though providers of ATM hardware will often implement a level of abstraction which provides an interface with a larger MTU, say 4KB or 8KB (FORE Systems, 1994)). Even in a shared memory system, it makes sense to operate on objects that are some multiple of the cache line or page size, depending on the granularity with which memory is shared. While we are unwilling to fix a particular MTU for all target architectures, we can parameterize the library routines by declaring that there will be a constant MTU whose value will be available at runtime.

There are a variety of benefits from assuming that the network will operate at its peak performance with messages that do not exceed a particular size, and the assumption can be applied usefully at all levels of the communication hierarchy. Many algorithms do not require that we bundle up a complete message, whose size may be data dependent, before sending part of it. For example, the first stage of a general get involves sending a request message to each node for each value that the remote node has and the local node requires. Sending a separate message for each value is very wasteful, and it is common practice to accumulate the requests to amortize the cost of communication (c.f. message vectorization, (Hiranandani, Kennedy, & Tseng, 1994)). However, if we were to wait until we had the list of all remote values the local node requires, we would require a buffer of nearly unbounded size for each remote node.

Though it is accepted wisdom (Balasundaram, Fox, Kennedy, & Kremer, 1991; Hiranandani *et al.*, 1994) that for small messages communications overhead is more closely related to the number, rather than the size, of the messages, this is not so clear when message size exceeds the network MTU. On many networks, the cost of sending a message of size $k \times$ MTU is roughly equal to the cost of sending k messages of size MTU, since in both cases the bottleneck of the transmission interface must be entered k times, and there is little if any benefit in allowing some intervening service to do the necessary fragmentation.[1] It is valuable to impose a reasonable limitation on the size of messages even when this is untrue, and transmission time is the cost of message startup plus a penalty directly proportional to message size no matter how large that may be. Without a limit, we would need to retain arbitrarily large buffers into which messages would be built, and ensure that these buffers were not improperly shared between functions that might be executing simultaneously through some sort of multiprogramming to increase efficiency. In addition to the space wastage for the accumulated request message, we would pay a penalty in latency by waiting until the whole message is ready before sending any of it, while we could instead overlap the local computation of what values are required from what nodes with at least some of the communication of the requests to the other nodes.

Even with an assumption that there is a preferred size for messages, we cannot refuse to transfer messages which exceed that size—for example, if a single element in a parallel value is larger than the MTU—and it would be undesirable to force every part of the system which might build a message for transmission to take responsibility for fragmenting the message to satisfy the MTU requirements. Therefore, the MTU is taken only as a hint, and we must retain the ability to transfer, when necessary, messages of whatever size might be required. (We may reasonably require that the MTU be at least some minimum size so that a common header along with some data may be transmitted; with the current system, the header will consume 16 bytes on a 32-bit machine, so useful MTUs must exceed this length.)

Related to this is the fact that most networks will limit the amount of data that can travel on the network at any time, or be buffered at a remote end. As long as processors can build messages faster than the network can deliver them, some sort of flow control will be required to ensure these buffer limitations are handled. The issue of flow control when a communication transmits a large amount of data does not seem to have been explicitly addressed by most researchers. Deadlock avoidance implies that, at least for cluster-based architectures, it is not sufficient to precede a loop with a single massive transmit, perform local operations, then read in data from other nodes. Unless one can assume the availability of asynchronous transmit and receive operations, deadlock avoidance mechanisms can be sufficiently complex that their integration into any communication optimization is non-trivial.

We do not wish to trouble higher level routines with flow control complexities, because the particular warning signs and recovery actions will vary depending on the underlying network. Flow control must be the responsibility of lower level routines, though we must be willing to provide some sort of mechanism, such as calling a check function at regular intervals, to give the low-level functions the opportunity to detect the situation and take compensatory action promptly.

1. For example, simple testing on Ethernet-connected Solaris workstations showed a performance loss of only 5% when a 32KB buffer was transmitted in 23 fragments of at most 1480 bytes rather than in one 32KB block.

We can also profitably take advantage of assumptions about the quality of service provided by a network. For example, TCP/IP will guarantee that messages are delivered exactly once, and in the order in which they were sent. On the other hand, UDP/IP may duplicate or drop packets, and messages may arrive out-of-order. Other protocols may guarantee that duplicates will not occur, but may fail to deliver a packet if some negotiated bandwidth limitation is exceeded by the program. Clearly, the type of failure is dependent on the underlying network, and it would be an implementation nightmare if the details of error recovery were able to propagate up into the higher levels of the communications hierarchy. Therefore, we will require that the lowest level will guarantee in-order reliable delivery of packets.

For some algorithms this is stronger than is strictly necessary—for example, it is not critical in the general get operation described above that requests for data be served in order, as long as all are eventually received (and this state is recognizable). However, there is often some benefit even in these algorithms in assuming that requests arrive in the order they were sent. State on the remote node, such as a current position in a data or context structure, as well as cache lines from previous requests, may require a sudden large shift if the order of requests received is not monotonic. However, many network interfaces support the assumed level of reliability, and even with those which do not the frequency with which the desired service level is not actually met can be fairly low (Mosberger, Turner, & Peterson, 1994) (though the results in this study, based on C* communications patterns on an FDDI token ring, conflict with an evaluation of C* communications patterns on Ethernet using UDP (Chandranmenon, Russell, & Hatcher, 1994)). The value of allowing for out-of-order reception of messages in the high-level protocols is therefore unclear.

Though some systems are intended to work on heterogeneous platforms (Skjellum, 1993; Weissman & Grimshaw, 1994; Crandall & Quinn, 1993), issues of byte re-ordering and differences in the representation of primitive data types such as C `int`s and `double`s are orthogonal to the basic communications structure. In general, the C* communications system is unaware of the internal structure of data that it is sending to other nodes (e.g., when transmitting parallel `struct`s with arbitrary fields), and therefore is unable to perform a translation to a common data representation. Rather than enter into the morass of structure inference or tagging data with type identifiers, we restrict our attention to homogeneous networks, which contain only one hardware or software architecture.

One final question is whether we should assume that all communications operations are point-to-point, or whether we can rely on the availability of some sort of multicast or broadcast operation to propagate information amongst all nodes quickly. Though many interfaces support one-to-many communications, there are often limitations on it: on standard Unix interfaces to Ethernet, broadcast may require running as a privileged user, while multicast is only available with a protocol that does not provide reliability guarantees, inducing extra overhead to meet the quality-of-service demands in the low-level routines. Furthermore, the most communication-intensive operations are inherently point-to-point: by number, very few operations in the C* library could take advantage of a broadcast mechanism. The hierarchy described in the next section is based on an assumption that only point-to-point message passing is supported. Experimental results which cast doubt on the value of using multicast even for the operations in which it might be expected to be beneficial, given our other system requirements, appear in section 4.4 after the details of the communications system are presented.

4.3 The Communications Hierarchy

The communications subsystem of pC* is structured into a hierarchy of three levels. The highest level routines implement core and library C* operations, such as grid and general communications, axial operations such as `spread` and `reduce`, and the more complex operations of `scan` and `rank`. These routines are almost completely divorced from any network limitations, except that they may where appropriate take advantage of knowing the network's MTU to decrease communication latency and buffering requirements.

The mid-level routines implement a generic set of communication functions and serve as an interface between the higher level routines and the lowest level, network-specific functions. These include:

- sending a value to a particular node
- reading from one (or any) node
- installing and invoking message handlers to decrease buffering requirements for incoming messages
- broadcast (each node contributes a portion of a whole which must eventually be available on all nodes)
- reduction (each node contributes an operand, the set of which are combined to yield a single value available on all nodes)

The interface presented accepts messages of any size, and performs all necessary fragmentation and network-independent buffering.

The lowest level routines have strong restrictions on what they can be asked to do, and in return guarantee to meet the reliability needs of the higher level routines. The mid-level routines ensure that low-level routines are not asked to handle messages that exceed the MTU, and that any incoming message is completely read before another is requested. In return, the low-level routines allow messages to be read piece-by-piece, so that header information can be used to determine where the remainder should be stored, and provide a mechanism which informs the mid-level routines that data from a particular node has arrived. This permits the mid-level routines to perform generic buffering operations to alleviate pressure on the network interface.

Figure 4.1 shows the hierarchy and dependencies between levels (solid lines) and routines within levels (dotted lines). Due to their complexity and number, not all high-level routines are depicted. Because communications functions are globally visible, the names are prefixed with `PCS__` to remove them from the user's namespace and avoid conflict with

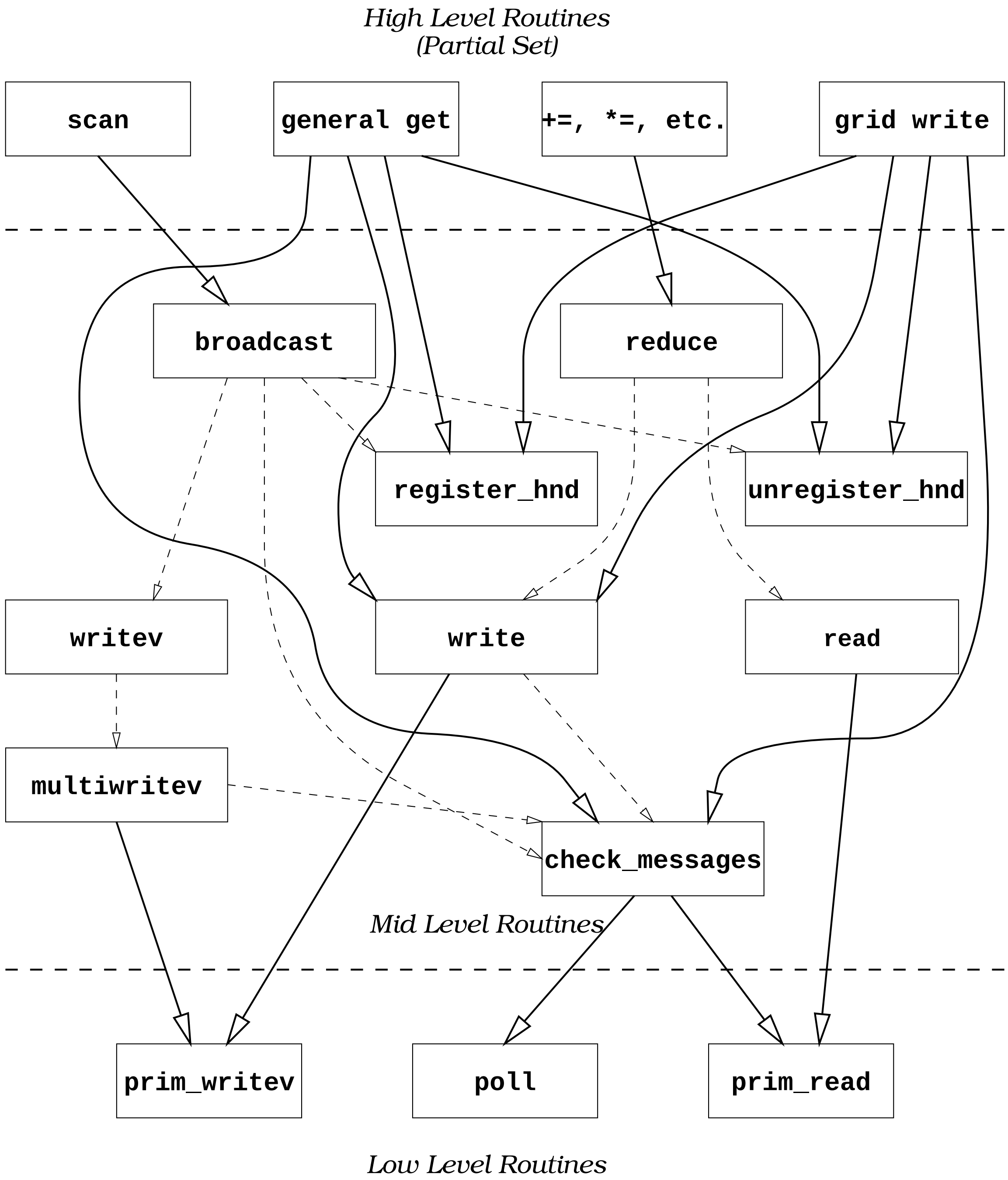


Figure 4.1: The Communications Hierarchy

system functions of the same name; this is reflected in the interface figures that follow, but the prefix is removed from the text descriptions to avoid unnecessary verbosity. A detailed description of each level follows.

4.3.1 Low-Level Communication Routines

Three routines are sufficient to meet the needs of the network-dependent component of the hierarchy; the interface is shown in figure 4.2. These provide the ability to write a message to a destination, read a message from a source, and determine what nodes have sent data. Nodes are named by their number within the cluster; where necessary, these must be mapped within the low-level routines to any network-specific information associated with the interface to the given node, such as IP address and port number, or Unix file descriptor.

`prim_writev` is the message transmission interface. The parameters to this procedure are the `partner`, or node number to which the message is to be written, an array of `prim_iovec` structures which point to the data to be sent, and the number of `prim_iovec` structures in the array. The `prim_iovec` structures are C `structs`, containing a pointer to a data block and an integral size indicating the amount of data to be sent from the block. This allows us to write to the network messages whose data are spread throughout memory; if the network does not support such a "gather" operation, the message can be packed inside the `prim_writev` implementation. It is assumed that there is a limit on the number of `prim_iovec` structures accepted by the network interface; this value is available to the system just as is the MTU.

Support for gather is extremely important, since every message which goes out must include a header, described in the section on mid-level routines, to identify the particular operation to which the message pertains. Similarly, at the higher level we may want to send a block of data from within a C* variable along with a structure containing instructions such as where on the remote node the data should be stored. Gather writes permit us to send directly from the original location of the data in memory, without unnecessary copying, when the interface allows this.

To aid in detecting network buffer overflows, the `prim_writev` routine is permitted to return without sending any of its message if it detects that there are insufficient resources to complete the message transmission. However, for correctness, it must guarantee that it sends either all or none of the message, lest the remote node read a header and attempt to go on under the assumption that the entire message is available. While some networks (such as fully-connected TCP socket meshes) keep all source/destination pairs separate, others rely on the header to determine the source of the incoming message, and will fail if messages are not transmitted atomically (as with named pipes where each node has a single incoming source through which all remote nodes send data to it). Failure to write a message is a signal to the mid-level routines that they should buffer any pending incoming messages to avoid network buffer overflow and deadlock as described in section 4.3.2. However, if the network interface permits writing a partial message, thus committing `prim_writev` to complete, the low-level routine may be obliged to take on some of this buffering role itself.

```
/* The structure used by the primitive readv and writev functions */
typedef struct PCSprim_iovec {
  void * iov_base;               /* Address of data block */
  int iov_len;                   /* Size of data block */
} PCSprim_iovec;

/* Primitive read function.  Read up to size bytes into buffer from
 * partner. */
int
PCS__prim_read (int partner,     /* Node number to read from */
                void * buffer,   /* Destination of data */
                int size);       /* Amount of data */

/* Write to the transport the combined data from the locations specified
 * by iov, as one message.  Guarantees no interleaving if total message
 * length does not exceed PCS__prim_msglimit. */
int
PCS__prim_writev (int partner,   /* Node number to write to */
                  PCSprim_iovec * iov, /* Specification of data sources */
                  int iovcnt  ,  /* Number of specifications */
                  int blockp);   /* Blocking/nonblocking write */

/* Generic polling function.  onodes is where the worker node number of
 * connections that satisfy the poll request are stored; nonodes says how
 * many there is room for.  Returns immediately if timeout is 0; waits
 * indefinitely if timeout is -1; waits timeout milliseconds if timeout is
 * positive.  Returns the number of entries filled in onodes; -1 if no
 * entries were filled and there was a non-zero timeout.  Errors are
 * fatal errors to the system. */
int
PCS__poll (unsigned int * onodes, /* Where output nodes should go */
           unsigned int nonodes, /* How many output nodes can we handle */
           int timeout);         /* Timeout on wait for event, msec */
```

Figure 4.2: Communications hierarchy: low-level interface

prim_read is the message reception interface. It is given the partner from which a message is to be read, the size of the data, and a pointer to the destination where the data should be stored. The read routine will block until the requested data has arrived, so care must be taken not to commit to a read unless it is known that the data will be available. It is not necessary to read the entire message at once; in fact, the preferred use of the prim_read routine is to read the header and use its contents to determine what should be done with the remainder of the message. This allows the "scatter" analog of the gather write supported by prim_writev, where we can store incoming data directly into its final location based on information provided by the header, saving both buffer space and copy operations. Though the mid-level routines may invoke prim_read multiple times to read a message, they guarantee that they will not attempt to read any other message until the partially consumed one has been completely read.

Finally, poll is used to check for the presence of incoming messages. The parameters include a onodes array to store the node numbers from which pending messages have arrived, a count nonodes indicating the maximum number of nodes that can be stored, and a timeout which permits us to control behavior when there are no pending messages: either to return immediately, or to wait up to the given timeout for an incoming message. This routine is used within the deadlock prevention scheme in the mid-level functions, and to determine the partner passed to prim_read.

These three routines make up the network-dependent components of the system, and a separate module (C source file compiling to a Unix object file) is maintained for each of the networks supported by the pC* system. Currently, five multiprocessor mechanisms are actively supported. These modules, averaging approximately 500 lines each, are maintained separately from the primary runtime library, and the user can specify at link time which network a particular C* program should use.

4.3.2 Mid-Level Communication Routines

The mid-level communication routines are more numerous, and significantly more complicated, than the low-level routines. They can themselves be partitioned into three sets: basic read and write, buffer handling, and collective communication routines.

All messages which pass through the mid-level are associated with a *transaction type*, which is an integer that indicates the type of operation with which the message is associated. There are approximately two dozen transaction types in the current system, most of which name phases in high-level routines; examples include owner broadcast, grid read, and general get request. Each type has an associated *index* number, which allows us to have several of the same transaction active at any time; the index number is incremented for each new operation. When combined the transaction type and index create a *transaction code*, an integer which uniquely associates a message with an operation. However, the mid-level routines are generally invoked with only the transaction type as a parameter; the code is constructed internally based on the current index.

The mid-level routines are responsible for fragmentation of data into MTU-sized pack-

```
typedef struct IPCmsghdr {
   int sender;              /* Node number of origin */
   int mcode;               /* Type tag, for synchronization */
   int size;                /* Size of the data portion of the msg */
   int index;               /* Index of message if split */
} IPCmsghdr;
```

Figure 4.3: Common message header

ets, and the corresponding reconstruction. To aid in reconstruction, each message passed to the low-level routines begins with a header whose structure is given in figure 4.3. In addition to the type code, information is provided about who sent the message, the size of the data portion of this message, and the index of the message when it is fragmented by the mid-level routines. The low-level routines may need to examine this header in order to determine who sent the message, or the appropriate order for returning message packets if in-order delivery is not guaranteed by the underlying network. If the low-level routines wish to piggy-back additional information into the messages they transmit, such as checksums or acknowledgements of previous messages, it is their responsibility to ensure this information does not propagate to the mid-level routines, which are not concerned with it.

Notice that the fields in the `IPCmsghdr` are all of type `int`. One might consider whether some space savings could be achieved by using types which are likely to be smaller. One reason for not doing so is the danger in limiting the ranges of the field, especially the `index` used in fragmentation. A 16 bit `index` field, in conjunction with a 512 byte MTU (reasonable for some interfaces, such as named pipes or shared memory machines), would limit single data transfers to 32MB. Though this sounds like a large message, we want the mid-level routines to behave as though there were no practical limit to message size, and in terms of large images 32MB would be a practical limitation. A more interesting reason to avoid the "non-native" word sizes is given in (Mosberger, Peterson, & O'Malley, 1995), which describes an analysis of TCP protocol latency on DEC Alpha workstations connected by Ethernet. The analysis found that the largest savings in the number of instructions executed to implement a protocol was due to using `int`s instead of `char`s or `short`s in state variables, because the Alpha does not have hardware support for accessing data in less than 32-bit chunks. As other architectures can also suffer from non-aligned memory accesses (in terms of speed if not additional instructions), the minimal space savings from packing structure fields was therefore not considered to be worthwhile.

4.3.2.1 Mid-level Write and Read

The mid-level routines shown in figure 4.4 include several methods of writing data to other nodes. The basic `write` routine is given a `destination` node, a message `type`, a `buffer` address, and a `size`, and sends the `size` bytes which live locally at `buffer` to the `destination` in as many MTU-sized messages as are required.

```
/* Read an arbitrarily sized block of data of code type from partner.
 * Handles interleaved messages and out-of-order delivery.  If size is -1,
 * reads until a  partial primitive packet has been read.  Returns total
 * number of bytes read */
int
PCS__read (int partner,          /* Who's sending the data? */
           MessageType type,     /* Message type code */
           void * buffer,        /* Where to store the data */
           int size);            /* Amount of data to expect. */

/* Send an arbitrary sized amount of data to partner, tagged as type, from
 * buffer.  Encases data in primitive-level packets as required by the
 * transport level.  The final packet will be partial, so PCS__read can
 * detect the end of an arbitrarily sized read.  This will send a null
 * message if size is 0. */
int
PCS__write (int partner,         /* Who to send to */
            MessageType type,    /* Message type tag */
            void * buffer,       /* Where data lives */
            int size);           /* Amount of data to send */

/* Package up the iovec along with any external data it references into a
 * single message, which is split as required by the network to avoid
 * interleave problems.  The message is sent to everybody in the dest
 * list. */
int
PCS__multiwritev (int * dests,  /* Who to send it to */
                  int ndests,   /* How many are there */
                  MessageType type, /* Type tag of message */
                  PCSiovec * iov, /* Data to send */
                  int iovcnt,   /* Number of iov elements. */
                  PCS__Bool sendiovcnt); /* Send iovcnt in message? */

/* Package up the iovec along with any external data it references into a
 * single message, which is split as required by the network to avoid
 * interleave problems.  The message is sent to the single partner
 * given. */
int
PCS__writev (int partner,       /* Who to send it to */
             MessageType type,  /* Type tag of message */
             PCSiovec * iov,    /* Data to send */
             int iovcnt,        /* Number of iov elements. */
             PCS__Bool sendiovcnt); /* Send iovcnt in message? */
```

Figure 4.4: Communications hierarchy: mid-level read/write interface

A somewhat more powerful routine is `writev`, which instead of the buffer and size arguments takes a pointer to a sequence of `PCSiovec` structures and a `count` of how many structures are in the sequence. These structures are similar to the ones used for the low-level interface, but include information about the source node (used in `broadcast` and high-level functions) and have the option of storing a scalar type code and a value (e.g., `int` or `float`) instead of a buffer size and external address. The network-level message that is formed from an invocation of `writev` consists of the sequence of `PCSiovec` structures followed by the data pointed to by members of the sequence which refer to memory blocks instead of containing scalar values. With the exception of the mid-level `broadcast` routine, all call sites to `writev` invoke it with only two `PCSiovec` structures, the first of which generally contains an integer which names an offset within an agreed-upon C* shape, and the second of which is a pointer to the local data which are to be transmitted to the remote node and stored in the given offset.

The third interface extends `writev` to a `multiwritev` function which, instead of sending the data to only one node, can send them to several or all the nodes in the computation. This permits higher level routines to assume that some sort of multicast one-to-many mechanism is available and saves repeatedly fragmenting the same message, even though the current implementation does not support broadcast directly. (In the code, `writev` is a front-end to `multiwritev`, so the fragmentation code is not duplicated.)

Incoming messages are received in one of two ways. Where synchronous transfer is required, a `read` routine is provided a `source` node, a message `type`, a `destination` address, and a `size`, and a message with the current code for that type is read from the `source` node and stored into the `destination`. `size` can be a wild-card indicating "the whole message, I don't know how big it is," or can be less than the whole message. In the latter case, `read` will go on to read and buffer locally any remaining portion of the current network packet, to maintain the invariant that partial packets are not left pending at the low-level interface. More commonly, incoming messages are handled asynchronously through an extension of the mechanism used to avoid network buffer limitations.

4.3.2.2 Mid-level Buffer Management

Recall our assumption that a given underlying network is likely to limit the amount of data that can be in transit or left unread. For example, experimentation under Sun's Solaris operating system indicates that stream sockets have an upper limit of approximately 40 KB pending data, named pipes a limit of 9 KB, and the System–V message passing facility of 4 KB. If the limit has been reached, the interface functions will either block or refuse to transmit more data (Papadopoulos & Parulkar, 1993). Though some of these limitations can be increased by modifying kernel variables, it is generally either impossible or unreasonable to extend them to the point where we can be sure the limitations will not be exceeded. Consider, then, what will happen if two nodes attempt to exchange large buffers, say 1 MB in size, during the same operation: each will start to fragment the buffer into MTU sized chunks and send them out, but soon the low-level `prim_writev` routine will indicate that it is unable to transfer the data because the other end is also busy writing instead of reading. Something must be done to alleviate the pressure on the network.

The problem is addressed with a cooperative buffer-management protocol. It is not possible to inform a remote node that we are no longer able to write to it.[2] Therefore, we must trust that it will notice this and deal with the problem itself. To avoid network buffer limitations, we check the interface (using the low-level `poll` routine) just prior to each `prim_writev`, and read in any messages which are pending at the interface. These are stored in buffer chains indexed by transaction code. When a write attempt fails, we pause and clean up our incoming messages, then attempt to retransmit. Since execution on all nodes is coupled (albeit loosely) by communication requirements, and all high-level communication operations have an implicit barrier which ensures that the operation has completed, we can be sure that the remote node will eventually check its incoming queue and remove enough pending material that we will be able to send more data to it. So long as the MTU is less than the network buffer limits, and the low-level routines atomically deliver or refuse to deliver messages in a bounded finite time, deadlock will be prevented.

It is not sufficient to check for pending messages only when we attempt to write, since the communications pattern for an operation might not cause write to be invoked with any regularity. Therefore, we provide a `check_messages` routine at the mid-level, which takes an optional timeout (described later) and reads in all pending messages and buffers them. This routine should be called at regular intervals in any portion of the code in which messages might be expected to arrive while we are not calling communications routines ourselves: for example, when walking the local portion of a shape in the grid communication routines (chapter 6), we can expect data from other nodes to arrive during the walk.

Though preemptive buffering of messages in this fashion avoids deadlock due to network buffer limitations, we have merely transferred the buffering requirements from the network interface into the mid-level communication module, which by design does not have any fixed limitations on how much data it will accept (as long as it can continue to allocate buffer memory). In the example above of exchanging two 1MB chunks, this means that after the transfer each side will have 1MB of data, buffered in chains of MTU-sized chunks, to walk through and deal with appropriately. It would be far better if we could, at the time we read the fragment, perform whatever operation was required immediately, thus avoiding the buffer at the mid-level (as well as the ensuing copy when it is moved to its final resting place). The message handler component of the mid-level hierarchy was designed to address this issue; it is somewhat analogous to the Active Message concept of (von Eicken, Culler, Goldstein, & Schauser, 1992), though more general and correspondingly of heavier weight.

Two functions complement `check_messages` by permitting it to detect expected messages and perform a specified operation on them (see figure 4.5). `register_handler` takes a transaction code, a pointer to a function, and a pointer to some arbitrary parameter, and records this information. When `check_messages` is informed through `poll` that

2. This may appear obvious; there are, however, interfaces which would permit transfer of out-of-band information such as error conditions even when the normal band is blocked. These are not universally available, hence cannot be relied upon in system design.

```
/* Type of function invoked on handled messages */
typedef void (* ChkMsgFunction) (
   unsigned int sender, /* Who sent it */
   unsigned int mcode,  /* Message code */
   void * msgbody,      /* Data in message */
   void * otherparm);   /* Parameter paired with mcode */

/* Look for any incoming messages that are sitting on the input ports:
 * we want to pull them off, because the other side might be blocking
 * on a write.  This function provides the ability to dispatch messages
 * that we expect, and buffer those we don't know what to do with. */
int
PCS__check_messages (int timeout);

/* Add a message handler: when a message with the given code appears
 * on an input channel, call the given function with the message as
 * a parameter. */
void
PCS__register_handler (unsigned int mcode, /* Code to look for */
                       ChkMsgFunction fn, /* Function to call */
                       void * params); /* Parameters to pass to fn */

/* Remove the handler associated with the given message code. */
void
PCS__unregister_handler (unsigned int mcode); /* Code to look for */
```

Figure 4.5: Communications hierarchy: mid-level buffer handler support

data are available from a given node, it reads in the header and compares the message's transaction code with the ones in its list. If no match is found, the remainder of the message is read and buffered. But if an entry for the incoming transaction code is discovered, `check_messages` reads the message body into a local buffer and invokes the provided function on the message.[3] `check_messages` is then free to avoid buffering this message, since the handler is presumed to have done whatever is required. The additional parameter is passed to the provided function along with message data, to allow the function access to other state such as variables into which incoming data should be stored or detailed information on the particular operation to be performed. A companion `unregister_handler` routine informs the mid-level that we have completed all operations corresponding to the given transaction code and removes the handler from the list examined by `check_messages`.

There are a variety of complexities involved in the message handling interface, from

3. Although this violates our desire to avoid copying buffers if the handler will simply move the message into place, most handler invocations, such as those from general and grid communications, need to pick values out of the message and store them in various different locations. In this case, no particular location is preferable for the message, and the overhead of asking the handler for a destination buffer is undesirable. Should our perception of this change, it is obvious how to extend the system to handle both cases (cf. the following discussion of `broadcast`).

trivial ones such as checking for buffered messages which arrived prior to the handler registration, to far more complex ones due to the fact that we do not restrict what operations may be performed by the handler function. As described in chapter 5, the handler function for a general get operation will read in a list of request values, which indicate what elements should be sent to a remote node. It will then package up those requests and send the data. The act of sending the data will itself invoke the `check_messages` routine, which may read in another request which arrived in the meantime. Were we to recurse into the handler for the new request immediately, we would trample on data being used by the previous, still active, invocation. Therefore, we must again buffer the incoming requests, being careful to maintain order of arrival, yet be sure that when the first `check_messages` call has finished it has dealt with not only messages pending at the network interface but also those which arrived and were buffered by recursive calls to `check_messages`. The issues of maintaining correct message order in a multiply–re-entrant handler make this one of the more complex routines in the system (based on the number of subtle, difficult-to-reproduce errors found during testing and production use).

By policy, we do not permit the two methods of reading messages to intermix in a given transaction type—messages which may be `read` may not have handlers installed for them, lest the complex buffer management required of handled messages be corrupted, or state maintained by the handler not be correctly updated. Therefore we use the `timeout` parameter to `check_messages` when we have completed all local operations and need to ensure that a transaction with a registered handler has completed. We assume that completion is noted by the handler on reception of the final message by setting some global state flag that can be examined in the high-level communication routine; we simply loop, at the end of the routine, calling `check_messages` until either the completion flag has been set or `check_messages` informs us that it has waited the expected time and has not received any additional messages. This permits us to distinguish between a remote node which is simply slow to respond and one which is no longer operational: such a failure results in a fatal runtime error, rather than permitting the program to block indefinitely.

It is possible that a highly imbalanced computation might result in an erroneous timeout if one node gives up before another has had a chance to complete its work. By default, the timeout is one minute for each node in the system (e.g., an 8-node cluster will wait up to 8 minutes for an expected message). Under normal processing in most applications, communication is common enough that nodes remain too closely coupled to exceed this timeout; however, if this should change, a more complex method of failure detection, perhaps based on a heartbeat system, could be implemented.

The message handler is used in almost all high-level communication operations, and the `broadcast` mid-level collective routine described in the next subsection. Instrumentation on the benchmark programs in chapter 7 running on eight nodes with medium-to-large sized tests indicates that 99% of messages intercepted by the `check_messages` routine are handled immediately, requiring no buffering (the lowest nodal handle rate was 90.9%), while the remainder were either unrecognized (arrived before the handler was installed), or required buffering to avoid out-of-order handling in nested calls to `check_messages`. The one benchmark that called a high-level routine that does not use the message handler needed to buffer half the messages it received, because they were for a different operation than the one the node was expecting. This implies that the message handler is saving a significant amount of buffer space, in addition to preventing deadlock.

Though we have said that most high-level operations have an implicit barrier at the end which indicates the communication has completed, the handler functions permit us to take advantage of data dependence analysis and move this barrier from the end of the high-level routine to a position in the code just before the result is to be used. For example, a general send operation may install a handler for incoming messages, send its data to remote nodes as appropriate, then continue processing without waiting for incoming data to arrive, saving the transaction code as a key. Just prior to use of the communication result, code can be generated to see whether the transaction has completed (based on the key which identifies it) and block until it has. Performing this operation would require a compiler analysis to determine that a particular parallel value results from a communication operation with a movable, handler-based barrier, and where the value will next be used. This analysis has not been implemented in pC*, so the optimization described here remains theoretical. However, it would appear that such an analysis would permit us to overlap the communications of data with local computation in almost every case, by making the compiler recognize values which are defined by operations with handlers, including any high-level library routine. We thus have a runtime system which, though designed without concern for compiler-based analysis that may or may not be available or applicable, integrates smoothly with such analysis when it can be done.

4.3.2.3 Mid-level Collective Communication

Collective communications (Mitra *et al.*, 1995) are interactions where a group of nodes cooperate to form a common result. There are two collective communications routines in the mid-level suite, which are used to communicate amongst a set of nodes rather than a pair. These are the `broadcast` and `reduce` operations, shown in figure 4.6. Though in both cases they involve communicating a contribution from each node, they have very different usage patterns, and the implementations are correspondingly dissimilar.

The purpose of `broadcast` is to take a block of data from each node, and ensure that after the broadcast every node has available to it the data from all nodes. The amount of data originating on each node may be different, and it is generally the case that the total amount of data is fairly large. For example, `broadcast` is used in the implementation of `read_from_pvar`, which transfers data from a distributed parallel value into a scalar array which is replicated on each node and contains all the values from the distributed value. Similarly, it is used for summary information when computing `scan` operations over all nodes, to indicate where scan set boundaries break the normal incremental progression along an axis. The size of data blocks in this case is proportional to the number of positions in the shape divided by the number of positions in the axis being scanned. In addition, the results of a broadcast operation are normally simply stored into some region of memory on each

```
/* An array of iovec structures, one per mesh node, to use for creating
 * and reading various broadcast data stuff */
extern PCSiovec * PCS__iovecvec; /* Data to be broadcast */

/* Function to locate buffer for broadcast data from src */
typedef void *
RecvDataFunc (unsigned int src, /* Node which sent the data */
              PCSiovec * iov,   /* Data received */
              void * params);   /* Parameters provided by caller */

/* Perform a broadcast involving the given nodes.  This node must
 * have filled out PCS__iovecvec[0] with its own data.  When the
 * function returns PCS__iovecvec will contain data from each node.
 * The handler function will have been called on the data from all
 * other nodes, but not this one.  Each node will have the same final
 * data in its own PCS__iovecvec, but probably in a different order;
 * the position of the data from this node may have changed.  */
void
PCS__broadcast (RecvDataFunc rfunc, /* Function to execute on receipt */
                void * rfparam,  /* specific params to pass to rfunc */
                unsigned int * bcmembers, /* Who's in the broadcast */
                int groupsize); /* Number of members in bcmembers  */

/* The data type for a function which performs a particular operation on
 * operands of known types. */
typedef void PCS__DoopFunction (void * lhsp, void * rhsp, size_t size);

/* General reduction operation, which applies doop to dest over all nodes.
 * Fans in down to first node, then returns result back, so everybody
 * agrees even if doop isn't associative. */
void
PCS__reduce (void * dest,       /* Our node's source / destination */
             int size,          /* Size of data being spread */
             PCS__DoopFunction * doop, /* Function to apply on receipt */
             unsigned int * members, /* List of members in reduction */
             int groupsize);     /* Number of members in reduction */
```

Figure 4.6: Communications hierarchy: mid-level collective communications routines

node, and are consulted unpredictably in further computation, rather than being condensed or operated on immediately as they are received.

The implementation of `broadcast` follows a fairly standard butterfly exchange algorithm (Mellor-Crummey & Scott, 1991) involving $\log P$ stages, since we have only point-to-point communications facilities available to us. Since the data involved are large, and we already have fragmentation support available, we call mid-level communications routines rather than low-level ones in the implementation. The `PCSiovec` facility of `writev` is used to allow us to read and write directly from and to the target buffers on each node. There is an array of `PCSiovec` structures, one for each node in the mesh, which is reserved for broadcast operations. In preparation, each node initializes the first `PCSiovec` structure with the information that it is contributing. The parameters to `broadcast` include a function which is called with the number of the originating node for a block of data when the first fragment of the data arrives, and which returns the memory address on the local machine into which the data is to be stored. This ensures that each node is able to arrange the data in a preferred order, without requiring special computation in the calling routine to determine the order in which blocks will be received during the log-based exchange.

To complete the exchange, each node first determines its partner for each stage, assigns a transaction code to the messages for the stage, and registers a handler to do the storing of incoming data. It also determines the number of elements in the global `PCSiovec` array which it must send to the partner, which contain the broadcast data that the partner has not yet seen. The array is managed so that at each stage an initial sequence of elements of the array must be sent: these consist of the data that originated on the node plus those which were sent to it in previous stages, allowing us to transmit data for a stage in a single call to `writev`. The handler routine associated with each stage will detect an incoming (fragmented) message, and store the data segments in the proper location, invoking the provided routine to determine buffer addresses as necessary: the handler is aware of the fragmentation algorithm used by `multiwritev` so it can reconstruct the data. When the handler detects that the last fragment of the message associated with a stage has completed, it calls the provided routine one more time to perform any associated clean-up code, and marks the stage finished. This permits the implementation to store data into their final location even if the data correspond to a stage which the local node has not yet reached, because it doesn't have what it needs to send in that stage. Since we are operating on large blocks of memory, this will avoid all buffering of data that arrive after the broadcast operation starts but before the data's stage is reached. The broadcast routine itself simply walks through the stages, sending the local data to the partner for that stage and waiting in `check_messages` for the necessary incoming data to arrive before proceeding to the next stage. At the completion of the operation, all nodes have all data, though the order of `PCSiovec` entries in the global array will be different on different nodes.

Pseudo-code for an implementation on a mesh with 2^k nodes is presented in figure 4.7. The handler routine `bcasthandler` is not shown; it simply unparses the fragments from `writev` and stores them appropriately. For meshes with $2^k < P < 2^{k+1}$ elements there are initial and final steps in which nodes n where $2^k \leq n$ send their data to a partner $n - 2^k$,

```
pmask = nnodes = 1;
sip = bcastsuminfo;  /* Array of summary info */
while (pmask < groupsize) {
  if (mynode & pmask) {  /* Chose our partner within mesh */
    sip->partner = mynode - pmask;
  } else {
    sip->partner = mynode + pmask;
  }
  sip->istart = nnodes;   /* Where we start storing iovecs */
  sip->nread = sip->nwrite = pmask;
  sip->state = BST_Initial;
  sip->mcode = NextMessageCode (MT_Broadcast);
  register_handler (sip->mcode, bcasthandler, sip);
  nnodes += sip->nread;   /* Number of data available for next stage */
  pmask <<= 1;
  sip++;
}
pmask = 1;
sip = bcastsuminfo;
while (pmask < groupsize) {
  /* Reset message index so writev uses correct code */
  SetMessageCode (MT_Broadcast, sip->mcode);
  writev (sip->partner, MT_Broadcast, PCS__iovecvec, sip->nwrite, 1);
  /* Wait until we've received everything for stage, or error */
  while (BST_Finished != sip->state) {
    if (0 > check_messages (io_timelimit)) {
      fatal ("broadcast: failed to complete stage %d\n", pmask);
    }
  }
  unregister_handler (sip->mcode);
  pmask <<= 1;
  ++sip;
}
```

Figure 4.7: Algorithm for Broadcast (Power-of-2 Mesh Case)

then the exchange is performed amongst the lowest 2^k nodes and the partner sends the final results back.

The `reduce` function differs in several ways from `broadcast`. First, the primary use is to generate a single scalar value which is a combination of the contributions from all nodes. The `reduce` function therefore takes a pointer to a function which, given two parameters d and s which point to memory regions, combines the data at s into d in an operation-specific fashion. The routine is most often used in C* reduction operations such as `s = += p`, where we are to store in the scalar `s` the sum of all active elements in the parallel value `p`. To implement this, each node computes the sum of the values in the region of `p` which it owns, then performs a global add reduction on the result.

One possible implementation is to use the `broadcast` operation to distribute each node's contribution, then walk the results locally computing the combined result. But the broadcast exchange results in a total of $O(P^2)$ network traffic, because it must transmit data from each of P nodes to each of $P-1$ other nodes. We could reduce this total traffic to $O(P)$ if we could perform the combination operation at each stage, thus receiving and sending only a single value. However, there is a potential difficulty with this.

It is critical for execution correctness that the results of a reduce operation be the same on all nodes, because scalar values determine control flow: a difference of only one bit can result in a divergence of execution on different nodes, with catastrophic results. If nodes receive reduction operands in different orders, this requirement may be violated. Depending on the reduction algorithm chosen, it may be sufficient that a reduction operator be commutative (i.e., $(a \oplus b) = (b \oplus a)$) or associative (i.e., $a \oplus (b \oplus c) = (a \oplus b) \oplus c$). However, associativity is not generally satisfied by arithmetic operations on floating point numbers, and commutativity is not satisfied by some other C* operations (e.g., parallel-to-scalar casts).

Since the amount of data communicated for reduce operations is generally very small—8 bytes or less—and will fit into the `PCSiovec` structure directly, we may simply choose to use `broadcast` without reducing in stages, since the amount of network traffic is small. We could then walk the data in a defined order based on source node (available in the global broadcast `PCSiovec` array) and perform the same sequence of combinations on each node, to yield a common final answer. The problem in this case is not correctness, but performance: the broadcast implementation was designed for transfer of large amounts of data, and the introduction of handler routines at the butterfly exchanges causes a significant overhead for such small messages. Comparing this method with the reduce algorithm described below on a variety of cluster sizes from 1 to 24 nodes, we found that the combination of additional network traffic and handler invocation caused, on average, a 15% performance penalty when performing reductions over 4-byte quantities; the difference was generally higher for non–power-of-2 meshes.

We would therefore prefer an alternative algorithm which maintains correctness but does not cause a performance penalty from unnecessary buffer handling. There are many possibilities, including simply writing the (small) data packets to all nodes and reading them in in order, or using log-based fan-ins to a single node which then distributes the answer to the other nodes in some fashion. Various algorithms were implemented and tested in the

```
pmask = 1;
while (pmask < groupsize) {
  if (0 == (mynode & (pmask-1))) {  /* Do I take part in stage? */
    if (mynode & pmask) { /* I'm source for this stage */
      write (mynode - pmask, MT_Reduction, dest, size);
    } else { /* Read from partner and combine locally */
      read (mynode - pmask, MT_Reduction, buffer, size);
      doop (dest, buffer, size);
    }
  }
  pmask <<= 1;
}
```

Figure 4.8: Algorithm for Reduce (Power-of-2 Mesh, Fan-In Phase)

process of comparing point-to-point with multicast support at the low-level communication routines; a discussion of the issues and results is given in section 4.4.

The algorithm finally chosen is the second best considered in section 4.4, due to a misapprehension about the correctness requirements for the best algorithm (cf. page 99). It uses a log-based fan-in operation to send contributions to a single node, which then performs the reverse fan-out operation to distribute the single result. Unlike the broadcast operation, at each stage a node either reads a value or writes one, but not both. Each node sends its value only once during the fan-in phase: after that point, it waits to receive the final result. Local combination is done at each read step of the fan-in, since the bottleneck of the single node ensures that there will be no disagreement on the answer. Since the data values are small and the fan-in/fan-out algorithms do not involve all nodes at all stages, we do not bother installing handlers to receive the incoming data, instead blocking on a read until the data arrive. (NB: the "blocking" mid-level read will detect and handle or buffer other messages which arrive in the meantime, so deadlock will not occur.) Pseudo-code for the fan-in phase appears in figure 4.8; the fan-out code is similar. For non–power-of-2 meshes initial and final phases similar to those described for `broadcast` send contributions and results to partners outside the primary fan-in group.

It is important to note that, in both these routines, the set of nodes over which the operation is performed is not necessarily the complete cluster: this is the purpose of the parameter which gives the set of nodes which cooperate. Synchroneity of control flow means that all nodes in the cluster will enter a particular collective communication routine at the same time. However, some invocations, such as the CSComm `reduce` function described in section 3.2.4, partition the cluster into groups based on ownership of data. For example, with a six-node cluster distributing a two-dimensional shape in both dimensions as shown in figure 3.2, a `reduce` along axis 0 involves two separate groups of nodes: nodes 0, 2, and 4 share information about the first two columns, while nodes 1, 3, and 5 share information about the last six. This separation means that in many cases true broadcast support would burden nodes by sending them information they do not need to know. Since there can be arbitrary partitioning of the cluster into broadcast/reduce groups depending on how the user has chosen to distribute data, it is necessary that we use a more insular implementation which ensures that data are distributed to exactly the nodes that require them. Of course, we must ensure that all nodes agree on the set of nodes that make up their broadcast group. A similar partitioning of clusters into node groups is recognized in other communications libraries (Barnett, Gupta, Payne, Shuler, Geijn, & Watts, 1994).

4.3.3 High-level Communication Routines

High-level routines in the communication hierarchy implement operations that are directly visible at the C* level, through core language operations such as left-indexed communication, parallel-to-scalar conversions, reductions, or communication-with-computation library calls such as `spread`, `scan`, `rank`, etc. The operations are often very complex, and we will not attempt to describe them in this section. Two particular examples of high-level communications routines are covered in other parts of this dissertation: the general communication functions are described in chapter 5, and grid communications operations are described in chapter 6. Both of these take advantage of various features of the communications hierarchy, including message handlers and the assumptions about MTU limitations, and are good examples of the integration of data layout, traversal, and communications decisions made in the pC* system. The other high-level communications routines, while interesting in their own right, do not provide fundamental insight to the issues that are the topic of this dissertation, and are left unexamined.

4.4 Point-to-Point or Multicast? A Case Study

Although pC* was designed to be portable to a variety of architectures, it was expected that for the first two years of development the primary target platform would be stock Unix-based workstations linked together with standard network hardware, using standard Unix system calls to perform communication. Under this environment there are two network interfaces which are obvious potential building blocks for the low-level routines of the communication hierarchy: stream sockets, generally implemented using TCP/IP, and datagram sockets, generally implemented using UDP/IP. Stream sockets offer reliable point-to-point delivery, while datagram sockets offer best-effort delivery and also provide access to a multicast facility.[4] We used stream sockets for the first year of development because they are reliable, but after some time it became clear that reduction operations were consuming a large portion of the execution time on some algorithms. For example, a common idiom for

4. Multicast support is distinguished from broadcast support by the fact that it can (a) be invoked by any user while broadcast requires superuser privileges (at least on common Unix/UDP implementations), and (b) restricts delivery to members of a group of (hardware) interfaces, rather than every interface on a connected subnet. While true broadcast might be useful in some applications, we prefer the safety of non-privileged execution and limiting reception of packets to machines which expect them.

iteration until convergence is:

```
do {
  old = latest;
  ... /* compute new latest */
} while (0 < += (old != latest));
```

which loops until the number of differences between successive iterations is zero. Though syntactically short, the add-reduction is a high-cost operation (often due more to the time it takes to communicate the reduction operands to all nodes than the time determining how many differences there are on the local node).

Although the quality-of-service of the datagram interface is not immediately adequate to meet our needs for a low-level implementation, common wisdom has it that communication over datagram sockets is significantly faster than over stream sockets. We wanted to know whether adding the necessary wrappers to UDP to meet the reliability requirements would yield a faster system, especially given that we could then (theoretically) take advantage of the multicast facility to improve reductions.

We performed two experiments to test this hypothesis. The first consisted of a program which simply timed a series of message exchanges between two processors, using a variety of message sizes and several different message passing protocols. This provided basic information on the performance of the different protocols over a range of MTUs. In the second experiment we examined the performance of two of the three core operations which could take direct advantage of a multicast mechanism: owner-broadcast, in which one node sends a single value to all nodes, and the mid-level reduce operation.[5] Several different algorithms were used to implement each operation, and their performance over the different message passing packages and message sizes was evaluated. In summary, the overhead induced when grafting our reliability requirements on top of UDP at user-level overwhelms any benefits from the faster underlying transport for point-to-point operations, and even when multicast operations are available their use should be considered carefully.

In all results described below, the experimental hardware was a set of twelve dual-processor Sun SPARC 20s running 60MHz SPARC chips with 256MB of memory, using Solaris 2.3, and connected in a star network through 10BaseT Ethernet with a Kalpana EtherSwitch EPS-2015 RS serving as hub. The primary purpose of the EtherSwitch is reducing network collisions due to multiple nodes sending data at the same time. It has been observed (LaRosa, 1995) that the synchronous nature of data-parallel computation often means that processors will enter communication phases at the same time. By putting each machine on its own leg of the star network with a switching hub at the center, point-to-point messages between nodes 0 and 1 (say) will not be visible to, hence not affect, any other nodes. Comparison tests done at the time the Kalpana was installed showed performance benefits on a wide variety of programs that ranged from a 10% slowdown to an 80% speedup, relative to

5. Mid-level broadcast was ignored, because some of the same algorithms that would be used for it were among those used to implement reduce, and conclusions about their performance on broadcast could be drawn from their performance on reduce.

a non-switching Ethernet hub. The bulk of the tests indicated a 20–50% speedup when six or more machines were involved.

Because UDP multicast is to hardware interfaces, not running processes, we could not use the dual processors to increase cluster sizes to 24, and tests were run with 2, 3, 4, 6, 8, 10, and 12 node clusters.[6] All programs were compiled using gcc 2.6.3 with optimizations `-O2 -msupersparc`, and during the tests no other non-system programs were running on the cluster. As a sanity check, the tests were also executed on a network of eight single-processor Sun IPCs running 25MHz SPARC chips with 24MB of memory each, also using Solaris 2.3 but connected on a standard 10Base2 Ethernet segment. The results observed in this environment followed the same trends as those described here, though performance tended to degrade more rapidly as cluster size grew, due to collisions on the Ethernet segment which were filtered out in the production cluster by using the EtherSwitch as a hub.

4.4.1 Ping-Pong Test

The first test program is similar to the Unix system utility `ping`(1).[7] A server process runs on one machine, awaiting connections. A client process connects to the server, and informs the server that it wishes to execute i iterations exchanging packets of size p. The time taken at the client end to perform the exchanges is recorded. The program can be compiled to use a variety of communication packages. Here we used the following four packages which implement an interface similar to the lowest level of the pC* communications hierarchy, except that any necessary polling is handled implicitly within the routines.

- Optimistic TCP opened stream sockets and used the Unix `read`(2) and `write`(2) system calls to exchange messages. We call this protocol "optimistic" because it makes no provision for recovery from or even detection of errors such as buffer overflow. No problems with buffer limitations were encountered during the experiment runs below (though limitations were exceeded when the packet size grew larger than is considered here). Only point-to-point between two nodes was supported, since this interface is only used in the ping-pong test.

- Optimistic UDP opened datagram sockets and used the Unix `read`(2) and `write`(2) system calls to exchange the messages. In the experiments below, the machines ran fast enough, and the total amount of data exchanged was small enough, that no packets were lost between the server and client, so no additional reliability support was implemented.

6. I.e., a multicast message is delivered once to the machine's Ethernet interface, and only one of the two processes would be able to read it. Supporting multiple processes on one machine would require forwarding the packet to the other processes through some other mechanism such as shared memory or inter-process messages, at a significant increase in code complexity. Other experiences indicate the second processor can have a small effect on execution time, either slightly improving it by off-loading work required to handle interrupts, or slightly degrading it by moving the running process between the two CPUs which have separate second-level 1MB caches. Nothing indicated that the cumulative effect was significant on these tests.

7. To distinguish standard functions from those of the pC* hierarchy, throughout this section we follow the Unix convention of marking, at their introduction, standard functions and programs with the section of the Unix manual in which they are described; hence (1) for programs and (2) for system calls.

- Reliable TCP is a small extension to TCP to support an arbitrary number of nodes in a communications mesh (a fully-connected graph of P nodes), and to support transaction codes, which are required in the second experiment to avoid mixing data between stages or iterations even when in-order delivery is guaranteed between node pairs. The write implementation interposes a function call which uses the gather `writev`(2) system call to send the message code, data size, and data buffer. The read implementation is somewhat more complicated, since it must `poll`(2) all sockets and buffer available data according to source and transaction code, to avoid deadlock due to buffer overflow or insufficient process read/write synchronization. To simulate a low-level interface which supports broadcast, there are separate functions to exchange point-to-point and broadcast messages; the implementation sets a bit in the transaction code to distinguish the nature of an incoming message. In this case, broadcast write is implemented as a series of $P-1$ point-to-point writes; broadcast read is identical to point-to-point read except for the queues on which it looks for previously-received data. The reliable TCP implementation consists of approximately 400 lines of C code, some 75–100 of which could be eliminated by merging the read functions.

- Reliable UDP is a major extension to UDP which, in addition to supporting transaction codes and buffering incoming messages, must also save outgoing messages until they are acknowledged, and retransmit them if the acknowledgement is not received in a "reasonable" period of time. Two datagram sockets are maintained on each node: the first is for reading and writing multicast messages, and the second is for reading and writing point-to-point messages and acknowledgements of both message kinds. On a write call of either kind, the header and data are packed into an allocated message buffer, which is written to the destination node and placed onto an "unacknowledged" list. On reception of a message, a 20-byte acknowledgement containing source node id, sequence number, transaction code, and flags is sent to the originating node. Incoming messages are sorted and buffered on point-to-point or broadcast queues as appropriate; out-of-order and duplicate delivery due to dropped messages are also handled. Message retransmission is performed if a message remains unacknowledged after 200 milliseconds; if only one node has failed to acknowledge a broadcast packet, the retransmission will be done along the point-to-point interface to avoid disturbing other nodes. Care was taken to use `poll`(2) to detect available messages and acknowledgement timeouts reasonably efficiently. The implementation is approximately 1300 lines of C code.

The last two implementations are sufficient to serve as communications primitives for the second experiment, though not quite adequate for a low-level facility in pC*: the reliable UDP implementation does not implement flow control, viz. there is no limit on the amount of unacknowledged data it will transmit and save. This was safe for the current application, which did not attempt to exceed the Ethernet MTU on any operation and had very close synchronization with almost no local processing, but would be inadequate for C* programs. Although both implementations include broadcast write and read functions, only the point-to-point interfaces were tested in the first experiment.

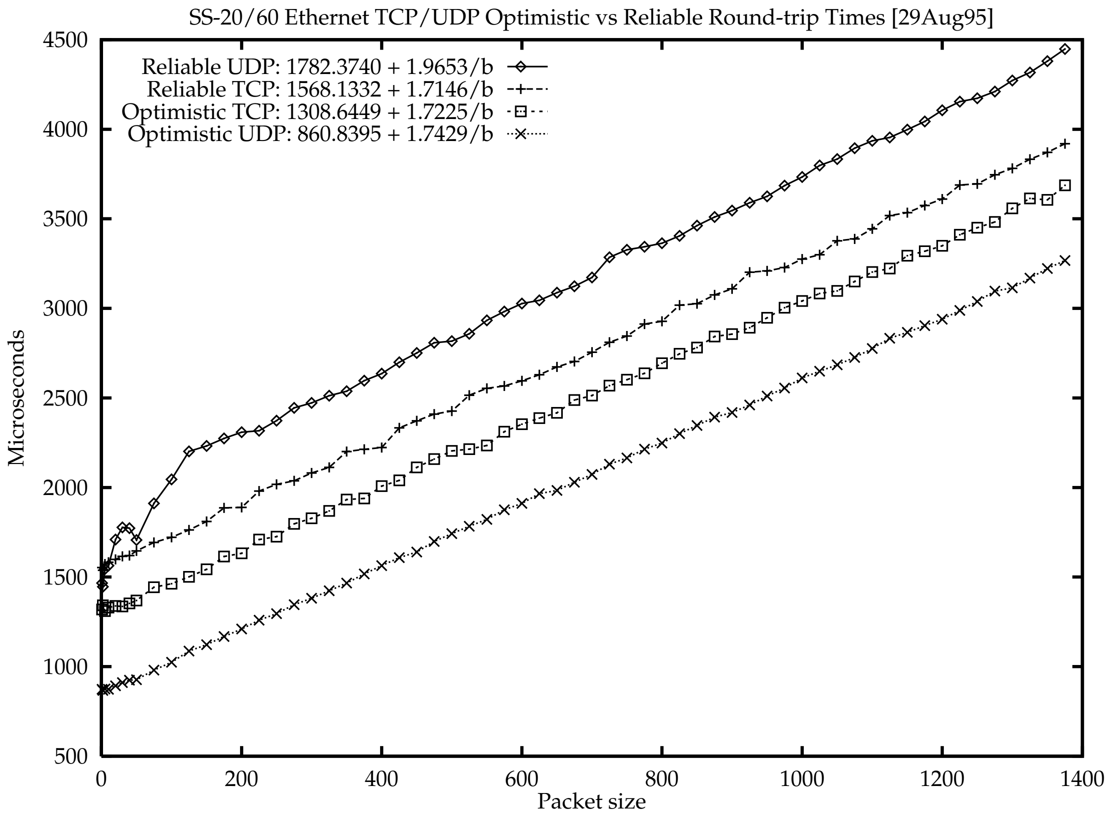


Figure 4.9: Ping-Pong Tests of Low-level Interfaces

The round trip times for each implementation are given in figure 4.9, along with a least-squares fitted linear approximation to give basic start-up and per-byte costs. The times that appear in the graph are the median of ten runs for each packet size, to abstract away from outliers due to network or processor interference, taken using a microsecond resolution real-time timer. Each run consisted of 100 packet exchanges; the final time was divided by 100 to yield the time in microseconds for a single [client write]–[server read]–[server write]–[client read] sequence.

The results for the optimistic protocols are as expected. UDP performs significantly better than TCP, with an overhead 448µsec (34%) less than TCP.[8] The per-byte cost for both is approximately 1.7µsec, only 0.1µsec (6%) higher than the lower bound of 1.6µsec due to 10Mbps transfer over 10BaseT Ethernet. The numbers are in concord with ones expectations for bidirectional exchange over Ethernet, implying the test is measuring what we want it to measure.

8. For reference, the Kalpana EtherSwitch introduces a 40µsec routing delay on all packets.

The reliable TCP implementation adds approximately 260µsec to the basic TCP implementation. Though we were unable to discretize the low-level implementation to determine exactly where the time is going, it is likely that the bulk of this time is due to the complex control flow around the `poll(2)` operations required to check incoming ports before reading data, plus overhead for checking buffer queues. Though the ping-pong test would never induce buffering or require checking more than one descriptor, these operations must be included in the general case implementation.

The biggest surprise comes with the reliable UDP implementation, which has an overhead 220µsec higher than that for the reliable TCP implementation, twice as large as the optimistic UDP implementation. Here there are many plausible culprits, including:

1. Copying the data from their original location into a buffer on write
2. Queueing the write buffer, regularly checking the queue for acknowledgement timeouts, and retransmitting when necessary
3. The complexity of polling to check for incoming data on both point-to-point and multicast sockets
4. Sending an acknowledgement for every data packet read and moving newly read data to the appropriate read queue
5. Unqueuing acknowledged packets

The above operations require multiple crossings of the user/kernel protection boundary to execute `read`, `write`, and `poll` system calls; measurements on separate programs indicate these system calls take between 15 and 30 µsec each for the smaller packet sizes (4–8 bytes). While we feel that the reliable UDP implementation has taken reasonable steps to ensure efficiency, it is clear that operations such as this, as well as some of the buffer management, would be better done inside the operating system kernel where domain crossing is not necessary and fewer copies need be made.

It is interesting to note that, unlike the other three interfaces, reliable UDP has a nonlinear performance curve for packets that are less than 125 bytes. A second sequence was run with a finer discretization of packet sizes, and is shown in figure 4.10. It confirms that the reliable UDP implementation starts out faster than the reliable TCP implementation, and crosses over somewhere around 20-byte payloads (the first to exceed 32 bytes message buffer size, including header). We hypothesize that this non-linearity is due to cache effects in copying data to and from larger buffers as the packet size increases. If a least-squares fit is taken for the reliable UDP data shown in figure 4.9 for packets at or above 125 bytes, we get an approximation of $1935.6269 + 1.8010/b$, which more accurately reflects the true start-up costs and brings the per-byte cost much closer to the expected value of processing time plus 1.6µsec for network transmission.

Naturally, further effort could improve the performance of the reliable UDP interface: imposing a reliable interface on an unreliable network fabric is a fundamental issue in

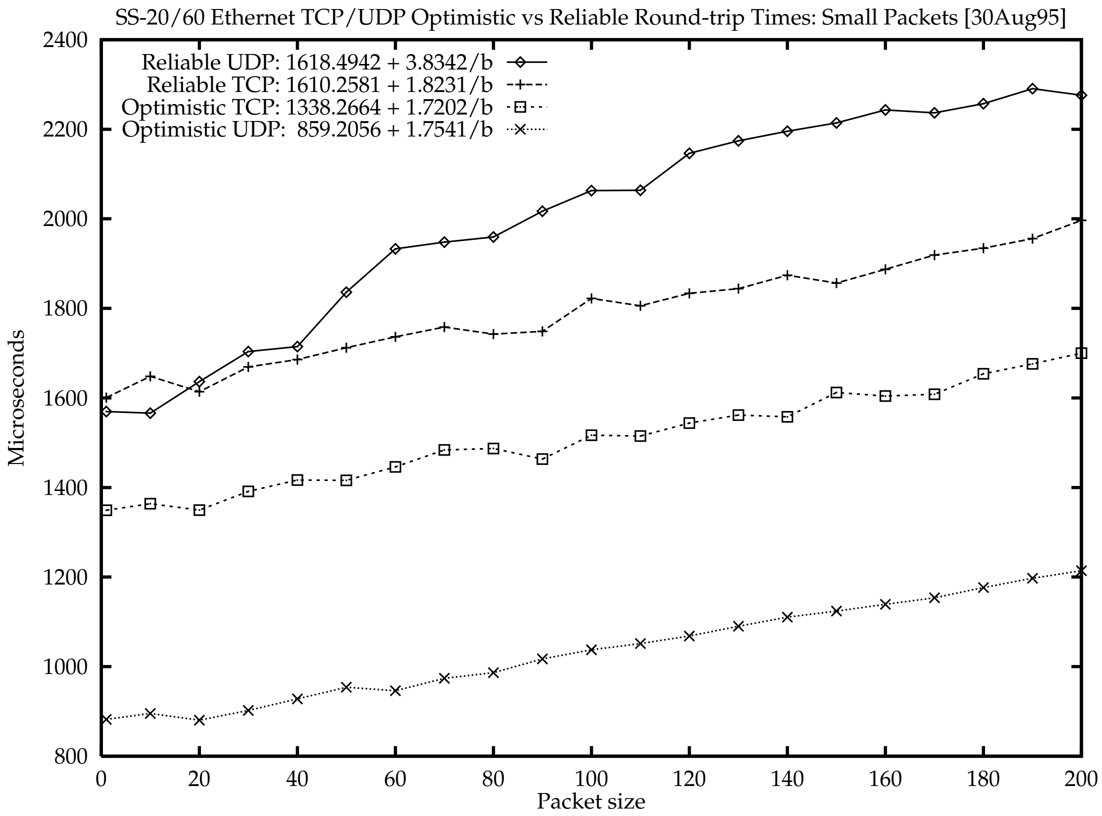


Figure 4.10: Ping-Pong Tests of Low-level Interfaces: Small Packets

network-based computing, and many issues such as timeout handling and flow control have a variety of solutions that can be judged based on intended use and the level of effort the implementor is willing to apply. However, we feel the implementation used here is defensible. Since we are precluded for reasons noted in section 4.2 from using certain features such as interrupt driven IO and interval timers, there are limits on how much improvement can be achieved by grafting on wrappers at the user level. For example, we do not attempt to piggyback acknowledgements on normal messages sent to remote nodes, because we do not know that an appropriate message will be sent soon enough, and cannot detect that enough time has passed that we should send an acknowledgement by itself. It seems unlikely that performance comparable to reliable TCP could be achieved, given the gap that must be closed and our implementation constraints. The only reason for going through the extra work, including extensions necessary to support flow control for exchanging multi-megabyte data blocks, would be a significant improvement in the C* algorithms which could take advantage of the multicast features. As we shall see, the improvement is not that impressive.

4.4.2 Broadcast and Reduction Algorithms

The second experiment consisted of a program which implemented two operations with a variety of algorithms, and linked to a low-level interface, similar to that described in section 4.3.1, which provided point-to-point and broadcast support. In the results presented here the reliable TCP and reliable UDP implementations of the previous section were used as the low-level interface. In each case, each algorithm was invoked in a loop of 100 iterations; the maximum time taken to execute the loop on any node was recorded. Each run of the program yielded one such time for each algorithm; the program was run five times for each cluster / data size pair, and the median number for each algorithm was recorded. We ran the above algorithms using both low-level interfaces on seven clusters from 2 to 12 nodes, and ten data sizes from 4 to 1400 bytes.[9] To conserve our nation's woodlands, we restrict the graphical results presented here to those for 4 byte and 1400 byte packets: most operations will be on scalar data such as `int`s or `float`s, which are represented by the 4-byte values, while the behavior on more rare large element types such as structures, or broadcast operations on large scanset summaries, can be inferred from the large packet results.

The first function to be implemented is C* owner broadcast: this is what is invoked when a scalar left index expression is used to distribute a single value to all nodes in the cluster, to be used as the value of a scalar expression. In this case, we have a single writer and $P-1$ readers. Three methods of performing this distribution were implemented:

MWRITE The source node invokes $P-1$ point-to-point write operations, sending the data to each of the nodes that require them.

LOGFAN A log-based fanout similar to the one used in the mid-level reduce function is used to distribute the value in $\lceil \log P \rceil$ stages of point-to-point transfers.

BCAST The low-level broadcast operation was invoked to transmit the data.

Because algorithms MWRITE and BCAST are not inherently synchronous, the time taken to execute them on the source node would be drastically different from the time on other nodes. To alleviate the extent to which this obscures the true performance of the algorithms, the iterations cycled the source node through the entire cluster.[10]

9. In the initial stages of developing pC*, we were unsure of the exact layout and payload of an Ethernet frame through various interfaces like TCP. We therefore chose a 1400 byte message limit (cf. section 4.2) as a safe upper bound for the socket system: this includes 16 bytes for the pC* IPC header, but does not include space for any headers added by the host system during transmission. This initial estimation remained in place until the late stages of preparing this dissertation, when instrumentation on the Solaris/TCP cluster indicated that the pC* message limit could be increased to a maximum of 1463 [*sic*] bytes before sending a message required multiple Ethernet frames. Testing done at that time showed no significant performance differences, on the benchmarks in chapter 7, between Solaris/TCP runs with 1400 and 1463-byte message limits.

Therefore, although later versions of pC* use the larger value, most performance results throughout this dissertation, including those of chapter 7, assume only 1400 bytes of user payload are available under a generic hosted Ethernet implementation.

10. This was not done in the synchronous LOGFAN, which always used node 0 as the source.

Graphs showing the execution time per operation over the cluster sizes for both interfaces and a hybrid to be described later are in figure 4.11. The time scales on the graphs are the same for each packet size, enabling visual comparison of performance between interfaces. For the point-to-point interface, BCAST and MWRITE are almost indistinguishable—as we would expect, since the broadcast operation in reliable TCP is implemented by multiple write statements. For both small and large packets the extra steps required for the log fan-out algorithm cause it to perform worse than the $P-1$ write operations. Performance for log-fanout is even worse using reliable UDP, where point-to-point messages are significantly more expensive. However, true multicast shows its value by performing in nearly constant time regardless of cluster size. At 12 nodes, the multicast owner-broadcast implementation runs 60% faster for 4-byte packets, and nearly five times faster for 1400-byte packets, than the multiwrite implementation. So for this function multicast support appears to be beneficial.

The second function to be tested is the mid-level reduce operation. Each node contributes a buffer of some fixed size; each algorithm tests a different communications pattern which would propagate the results to all nodes allowing application of a binary function to them (though in fact no such function was used, since we were interested only in communication times). Eight implementations of this operation were tested:

NAIVEMW Each node invoked $P-1$ write operations to send its data to the other nodes. It then looped reading $P-1$ statements from other nodes, in the order they arrived (using a wildcard source node).

MASTERMW Each node other than node 0 wrote its data to node 0, which performed the necessary combination and wrote the answer back to the rest of the cluster using $P-1$ write operations.

MASTERBC Each node other than node 0 wrote its data to node 0, which performed the necessary combination and wrote the answer back to the rest of the cluster using a broadcast operation.

REALBC The low-level broadcast operation was used to transmit each node's data to all other nodes, then the broadcast read operation was invoked to read the $P-1$ remote values in arbitrary order.

LOGLOCEX A butterfly implementation to exchange data with all other nodes in $\lceil \log P \rceil$ stages (a log exchange with local calculation of the final result).

LOGMW A log fan-in to node 0, which then used $P-1$ point-to-point write operations to send the final result to the other nodes.

LOGLOG A log fan-in to node 0, with a corresponding log fan-out back to the remainder of the cluster: this is the algorithm described in section 4.3.2.

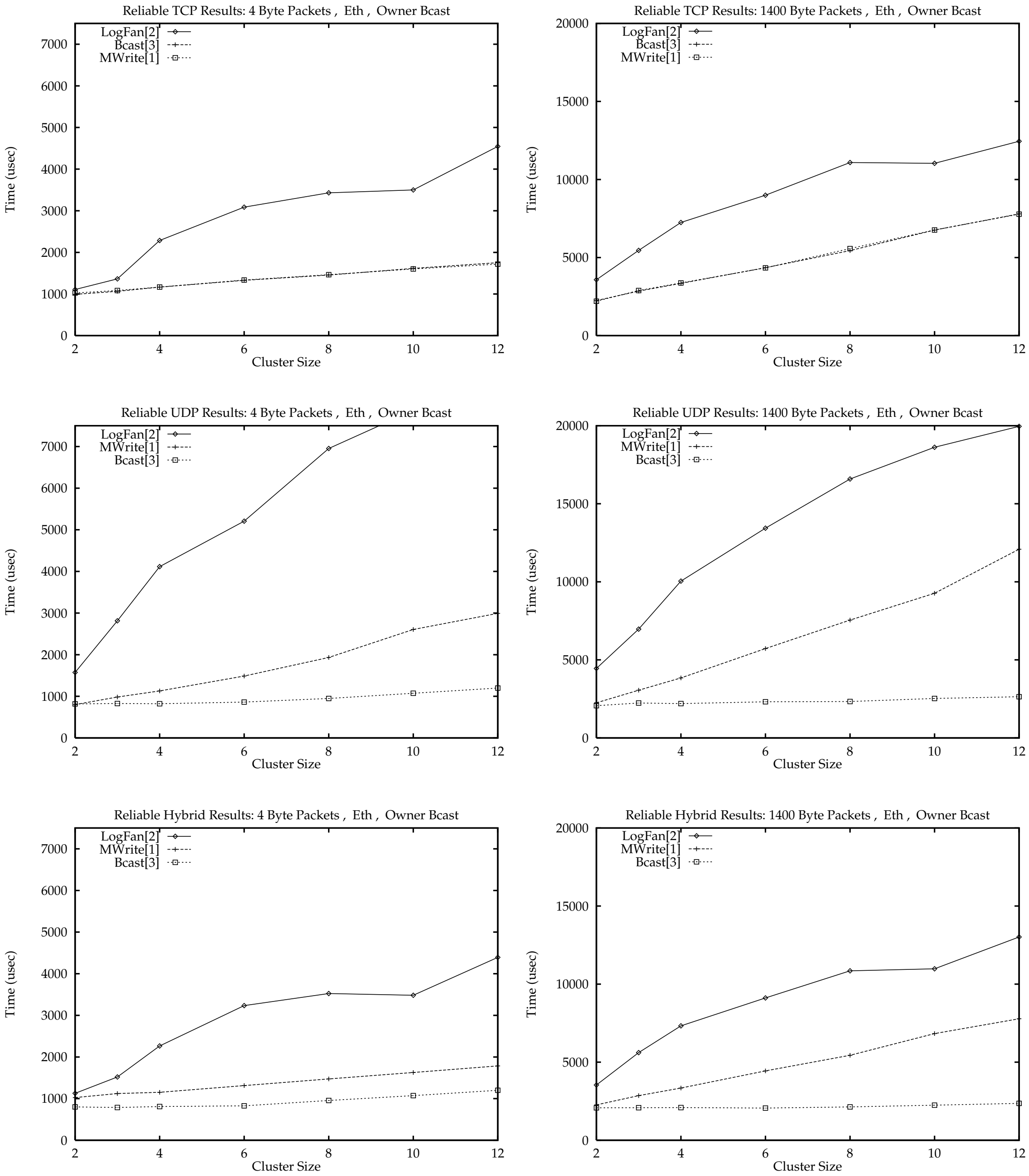


Figure 4.11: Owner-Broadcast Function Algorithm Comparisons

LOGBC A log fan-in to node 0, which used the broadcast write operation to send the result to the remainder of the cluster.

Since reduction operations are synchronous by nature, we did not vary the master node in MASTERMW and MASTERBC as we did with the corresponding owner-broadcast algorithms. Note also that NAIVEMW and REALBC do not satisfy correctness requirements for non-associative operators, though they could be modified to do so. LOGLOCEX can yield an incorrect answer with a non-commutative operator.

Graphs showing the performance of these algorithms under the various interfaces are in figure 4.12. Entries in the legends are sorted by decreasing runtime at 12 nodes. The time range on the *Y* axis was chosen to give a good overview of the relative performance of most algorithms, even when one or two algorithms performed significantly worse than the upper limit used. For example, the LOGMW and LOGBC operations with the point-to-point interface (both of which have identical communication behavior due to emulation of broadcast) suddenly jump to 50 milliseconds per iteration as soon as the cluster size reaches four nodes, and remain at 50msec up to 12 nodes. This appears to be due to an inefficiency in Solaris 2.3 TCP code with this particular communications pattern, since similar behavior was not seen in the UDP implementation. It does not occur with the MASTERMW or NAIVEMW algorithms, so is not due solely to attempting to write $P-1$ packets to different nodes in a very short time.

For reliable TCP, the log based algorithms that do not invoke a multiwrite operation have a flatter performance profile for the range of packet sizes, though they are slightly slower than the other implementations for small packets on small clusters, becoming fastest when cluster size exceeds 8 nodes. The log-based methods show their superiority on large packets where they cause significantly less data to be transmitted than any of the multiwrite-based algorithms; for smaller packets the difference between algorithms is much smaller.

The performance using reliable UDP with multicast support is much more intriguing. The REALBC implementation using multicast from each node has the second worst performance with 6 or more nodes and small packets, coming in at over four times slower than LOGBC for 12 node clusters (only two times slower for 1400-byte packets). Again log-based fan-in algorithms prove to be the fastest, though use of broadcast by the bottleneck node to distribute the result proves to be valuable.

These results imply that the most effective use of multicast is in a supplementary role, implementing a one-to-many rather than a many-to-many communications pattern. Though LOGBC is fastest in the reliable UDP implementation, it is still 53% slower than the reliable TCP implementation of LOGLOCEX on small packets. The results of the ping-pong test imply that this is due to the high overhead of reliable UDP on point-to-point messages, which make up the first phase of the LOGBC algorithm. Therefore we constructed a hybrid interface combining the reliable TCP and UDP implementations, where the TCP implementation was used for all point-to-point operations, and UDP restricted to multicast operations. The results of using this interface on the same algorithms are also shown in figures 4.11 and 4.12; it is clear that the algorithms which depend highly on point-to-point communications

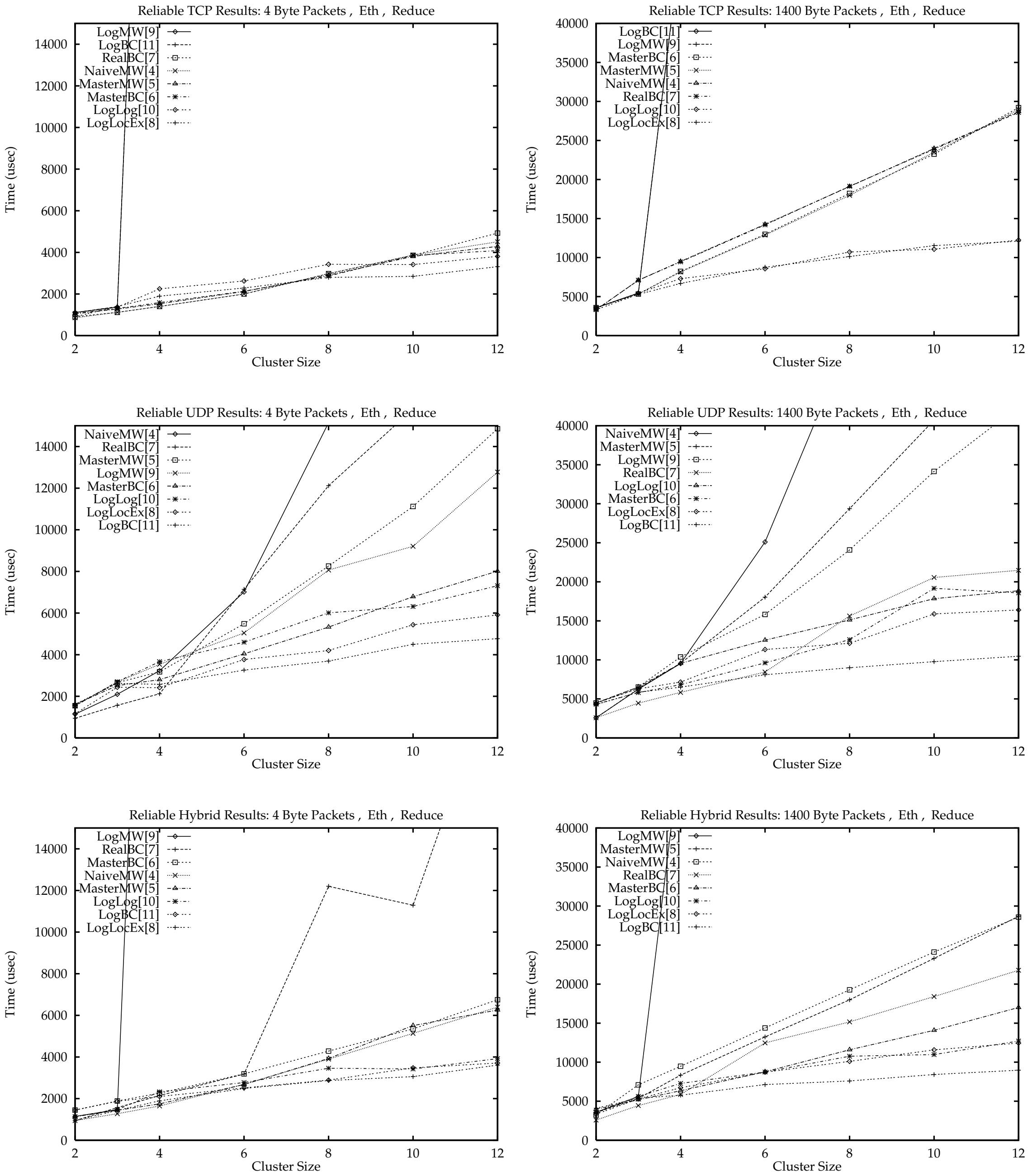


Figure 4.12: Reduce Function Algorithm Comparisons

retain their reliable TCP performance, while those which use broadcast follow the reliable UDP performance curves. The performance of LOGBC using the hybrid relative to the UDP interface improves 21% on 4-byte packets and 11% on 1400-byte packets through use of the more efficient point-to-point communications, and remains the fastest method for all packet sizes (though it is closely challenged by LOGLOCEX).

4.4.3 Evaluation and Conclusions

We have collected a large amount of data that compares two implementations of low-level communications operations, plus a hybrid that combines their best features. We must now evaluate the results and determine which implementation is most useful, not merely for the broadcast and reduce operations tested here but for pC* as a whole.

There are two axes along which the performance of low-level communication implementations vary: the size of the cluster, and the size of the data packet. Evaluation of algorithms for owner-broadcast is relatively simple: the multicast features of reliable UDP uniformly beat the point-to-point implementations required with reliable TCP, with smallest improvement 6% faster at 1400-byte packets with 2 nodes, and largest improvement 66% faster at 1400-byte packets with 12 nodes. The improvement on 4-byte packets ranges from 20% to 36%. We must consider this in context with the frequency of use of owner-broadcast, however. It is common in some numerical analysis algorithms (e.g., it would be used to broadcast the selected pivot element in LU-decomposition), but is relatively rare in image processing algorithms, and in almost all cases only small single scalar elements such as `ints` or `floats` would be distributed.

Reduction operations are more common and, unfortunately, the performance issues are more complex. The major contenders within the reliable TCP implementation are LOGLOCEX, LOGLOG, and NAIVEMW. LOGLOG is slower than LOGLOCEX almost uniformly, but improves as packet size increases. For small packets NAIVEMW (or, equivalently for this implementation, REALBC) performs better but it degrades as packet and cluster sizes increase. The trends exhibited by the graphs indicate that the log-based routines will continue to perform well on larger clusters, while NAIVEMW does not scale as well. See tables 4.1 and 4.2 for the performance of LOGLOG and NAIVEMW respectively, relative to LOGLOCEX.

We can reject the pure reliable UDP implementation because of its poor performance on the point-to-point communications which make up the vast majority of C* communications outside the operations of owner broadcast and reduce. Within the hybrid scheme, the top three contenders over all cluster and packet sizes are LOGBC, LOGLOCEX, and LOGLOG. It is interesting to note that, even with the improved performance by using TCP for point-to-point communications, LOGBC is still slightly slower than the pure log-based method, though it does do better on larger packets and cluster sizes. The performance of LOGBC and LOGLOG relative to LOGLOCEX on all cluster and packet sizes is given in tables 4.3 and 4.4, respectively.

Having narrowed the field, and with no obvious reason to prefer an algorithm which

Packet Size	Cluster Size 2	3	4	6	8	10	12	Geometric Mean
4	1.225	0.979	1.185	1.147	1.227	1.198	1.149	1.156
8	1.203	0.994	1.171	1.113	1.134	1.219	1.172	1.141
16	1.302	1.003	1.234	1.146	1.221	1.219	1.145	1.178
32	1.358	0.989	1.201	1.137	1.251	1.246	1.101	1.178
64	1.191	1.008	1.161	1.145	1.201	1.206	1.110	1.144
128	1.196	1.023	1.173	1.089	1.177	1.200	1.179	1.146
256	1.149	1.035	1.157	1.082	1.145	1.080	1.187	1.118
512	1.146	1.050	1.112	1.120	1.109	1.125	1.104	1.109
1024	1.092	1.036	1.079	0.999	1.056	0.982	1.031	1.039
1400	1.076	1.030	1.092	0.977	1.057	0.960	1.012	1.028
GMean	1.191	1.014	1.155	1.094	1.156	1.139	1.118	1.123

Table 4.1: Reliable TCP: LOGLOG relative to LOGLOCEX

Packet Size	Cluster Size 2	3	4	6	8	10	12	Geometric Mean
4	0.948	0.799	0.739	0.875	1.067	1.358	1.361	0.995
8	0.943	0.802	0.726	0.848	0.959	1.373	1.505	0.989
16	1.008	0.749	0.748	0.845	0.913	1.230	1.374	0.957
32	1.007	0.763	0.726	0.860	1.054	1.188	1.312	0.967
64	0.935	0.855	0.731	0.880	0.973	1.289	1.309	0.976
128	0.992	0.857	0.740	0.947	1.003	1.316	1.406	1.015
256	0.946	0.918	0.824	1.064	1.171	1.428	1.601	1.107
512	1.000	1.132	1.000	1.323	1.393	1.727	1.987	1.325
1024	0.997	1.305	1.330	1.525	1.729	2.013	2.248	1.541
1400	0.996	1.343	1.420	1.626	1.885	2.078	2.372	1.615
GMean	0.977	0.931	0.869	1.046	1.177	1.472	1.608	1.127

Table 4.2: Reliable TCP: NAIVEMW relative to LOGLOCEX

Packet Size	Cluster Size 2	3	4	6	8	10	12	Geometric Mean
4	1.530	1.304	1.110	1.012	1.007	1.137	1.027	1.148
8	1.542	1.161	1.125	1.026	0.977	1.097	1.050	1.128
16	1.615	1.242	1.078	1.002	0.993	1.081	1.066	1.138
32	1.662	1.172	1.093	1.022	0.890	0.999	1.032	1.104
64	1.487	1.236	1.026	0.979	0.946	1.121	0.984	1.098
128	1.579	1.147	1.023	0.972	0.855	1.000	0.865	1.042
256	1.399	1.207	0.949	0.929	0.835	0.952	0.868	1.004
512	1.302	1.188	0.901	0.894	0.759	0.830	0.780	0.932
1024	1.267	1.081	0.870	0.870	0.739	0.738	0.757	0.886
1400	1.230	1.013	0.859	0.818	0.751	0.727	0.721	0.859
GMean	1.454	1.172	0.999	0.950	0.870	0.956	0.906	1.029

Table 4.3: Reliable Hybrid: LOGBC relative to LOGLOGEX

Packet Size	Cluster Size 2	3	4	6	8	10	12	Geometric Mean
4	1.205	1.026	1.229	1.115	1.206	1.120	1.086	1.139
8	1.262	1.001	1.260	1.097	1.199	1.135	1.139	1.153
16	1.211	1.000	1.196	1.182	1.207	1.098	1.152	1.147
32	1.286	1.000	1.217	1.124	1.094	1.052	1.090	1.120
64	1.185	1.034	1.165	1.092	1.166	1.157	1.157	1.135
128	1.252	0.998	1.169	1.148	1.124	1.146	1.025	1.120
256	1.168	1.058	1.114	1.087	1.148	1.110	1.040	1.103
512	1.096	1.113	1.089	1.125	1.066	1.135	1.114	1.105
1024	1.121	1.077	1.061	1.030	1.077	0.973	1.070	1.057
1400	1.077	1.058	1.083	1.002	1.064	0.947	1.021	1.035
GMean	1.184	1.036	1.156	1.099	1.134	1.085	1.088	1.111

Table 4.4: Reliable Hybrid: LOGLOG relative to LOGLOCEX

Packet Size	Cluster Size 2	3	4	6	8	10	12	Geometric Mean
4	1.599	1.338	1.109	1.099	1.034	1.220	1.120	1.205
8	1.605	1.385	1.085	1.107	0.922	1.127	1.116	1.175
16	1.684	1.253	1.089	1.084	1.025	1.200	1.118	1.192
32	1.705	1.284	1.099	1.112	0.997	1.123	1.095	1.185
64	1.533	1.353	1.038	1.114	0.966	1.179	1.046	1.162
128	1.604	1.358	1.046	1.038	0.896	1.036	0.974	1.115
256	1.404	1.244	0.978	1.004	0.864	0.959	0.975	1.048
512	1.357	1.156	0.915	0.919	0.775	0.805	0.809	0.943
1024	1.222	1.087	0.885	0.849	0.744	0.747	0.762	0.884
1400	1.233	1.016	0.863	0.813	0.749	0.730	0.742	0.863
GMean	1.485	1.242	1.007	1.008	0.891	0.995	0.964	1.069

Table 4.5: Reliable Hybrid LOGBC relative to Reliable TCP LOGLOCEX

comes in second or third overall, we can now make a choice between using LOGLOG with the reliable TCP interface and using LOGBC with the hybrid interface. The relative performance numbers are given in table 4.5. On average, using hardware multicast does not give us a speedup over the point-to-point algorithm. This is true across all clusters for small packets. However, we would get a benefit, peaking at about 25%, by using the hardware multicast algorithm on large packets and clusters.

Qualitatively we must weigh these results against the following mitigating factors.

- The only time using the multicast support could provide an improvement is in owner-broadcast, reduce, and broadcast operations. Far more frequent in terms of amount of data sent across the network are general communication and grid communication, neither of which would benefit from multicast, and complex computation-plus-communication functions such as `scan` and `spread` which could only benefit if data distributions were restricted to one axis, so that all nodes participated in each reduction group.[11]

- It is specifically communications latency and the fact that the implementation blocks until data arrive which causes bad performance on reductions on small data. Using the compiler analysis suggested in section 4.3.2 to move the synchronization operation up to the point where the reduced value is needed would allow us to perform other useful computations during this latency time (though we would add overhead by using the mes-

11. Counting static calls to communication routines in the benchmarks described in chapter 7, we find fourteen are for grid communications, nine for general communications, three for other complex communications, and only five for reduce or broadcast-type operations. Of those five, two would require the single-axis-distribution restriction to allow hardware broadcast to be used. In the benchmarks examined there are more general and grid communications calls in loops than there are reduction or broadcast operations, so dynamic frequency of operations that can make use of multicast support is even less.

sage handler facilities on small messages). This implies implementation effort would be better directed to this analysis, which would improve other operations as well.

- The source for the hybrid interface is roughly four times as long as that for the TCP system. Its internal complexity is also much higher, with more buffering and polling being required. This makes correctness verification and maintenance significantly more complex.

- Neither multicast implementation has been tested inside the pC* system, where communications often involve data blocks that are much larger than the network MTU, nor with other communications patterns. Experience writing the TCP-based low-level module for pC* indicates that there can be issues that will only arise in such a communication-intensive environment.

- The current primary production platform is networked multiprocessors. As noted previously, supporting multicast with multiple processes on a single network node would require internal forwarding of packets. Given the small improvement in performance, there is a high probability that the additional overhead will overcome the benefits of supporting multicast.

- While the current TCP implementation in pC* is relatively simple and hence should be portable, a more complex UDP implementation, especially bound to a (most likely platform-specific) inter-process forwarding mechanism, is far less likely to be usable on a different target platform.

Taking all these issues into account, we have chosen not to attempt to use any UDP-based communications in the current pC* system, and all performance results in the remainder of this dissertation rely on other low-level interface modules. The decision is based on our own understanding of the goals and tradeoffs relative to our own application; we hope that the experiences related here will prove useful to other researchers faced with similar choices in their own systems.

4.4.3.1 Why pC* Didn't Use the Best Algorithm

Recall that the LOGLOCEX algorithm considered in this section does not satisfy correctness requirements when the operator is not commutative. This was originally discovered in pC* when a particular operation—parallel-to-scalar cast, which selects an arbitrary active element of a pvar and returns it as a scalar—caused problems because nodes disagreed on the chosen value. We initially assumed that the disagreement came from the non-associativity of the operator, and therefore that LOGLOCEX was unacceptable as a general-purpose reduction algorithm because floating point arithmetic is non-associative.[12] As a result, we chose to use LOGLOG as the reduction algorithm, and the experimental results in chapter 7

12. This is due to the fixed-point nature of most hardware implementations of floating point arithmetic, expressed as $0.m \times 2^e$ where m and e are integers represented with a fixed number of bits. Consider forming

use this algorithm. The comparison in table 4.1 shows that using the sub-optimal algorithm could cost us about 12% on average, and is likely to be responsible for some of the problems we encountered relative to the performance of reduction on power-of-2 sized meshes (note that 2-, 4-, and 8-processor clusters are generally much slower relative to LOGLOCEX than the other sizes). We have since realized that LOGLOCEX is perfectly acceptable to non-associative operators because it combines values in pairs at each stage, where at most the left and right operands are swapped. The few non-commutative operators used in pC* have been modified to enforce an order on their operands, making them commutative, and we now use the LOGLOCEX algorithm instead of the LOGLOG one described in section 4.3.2.

4.5 Conclusions

This chapter has presented an overview of the issues involved in choosing a communications infrastructure on which to build a runtime system for a distributed language. The trade-offs along the spectrum from portability to performance were examined, and a middle ground chosen which embodies a three-level hierarchy to isolate target-specific code from the routines invoked by the C* programmer. At the highest level, language-specific functions are coded using full knowledge of the expected behavior of the runtime system, without much concern for the details of the actual communications network. At the lowest level, we must provide a very few functions with clearly delimited rights and responsibilities, making it relatively simple to add support for a new network interface. An intervening level links the two, providing an essentially unconstrained communications interface to the highest level while ensuring the lowest level need not be burdened with message fragmentation or buffering which are common to all possible platforms for distributed computing. The next two chapters will examine in detail high-level functions which use this hierarchy to good effect.

This hierarchy has resulted in reasonable performance (to be shown in chapter 7) and proven portability, with five underlying networks currently supported, ranging from TCP sockets over a cluster of workstations to shared and distributed memory machines like the Sequent Symmetry and Intel Paragon. Providing definitions for the three low-level functions on a new platform is relatively straightforward, taking on average one programmer-day each for the last three modules implemented (none of which required extensive enhancements for reliability).

Though portability is our second goal in this project (reliability over the range of potential applications taking precedence), the system has still been designed to integrate with advanced techniques to improve communications performance, such as better management of buffers (supported through the scatter/gather interfaces and message handlers) and direct control of the network system. Integration has not been proven, and with some thought it

the sum of a, b, and c, where $(a,b) \ll c$. If evaluation proceeds as $a+(b+c)$, and a and b are smaller than $0.00\ldots01 \times 2^e$, then $b+c=c$ and $a+c=c$, so the final result is c. However, if a and b are large enough that $a+b$ is at least $0.00\ldots01 \times 2^e$, then $(a+b)+c \neq c$.

is clear that slight changes may be necessary. For example, on some architectures performance could be improved by using asynchronous writes where the low-level routines queue a transmission (perhaps through a separate DMA controller) but data must remain in their original buffer until the transmission is complete. Since some high-level routines place their data directly into MTU-sized buffers, provision must be made to tell the high-level routine to switch to a new buffer until the previous one is again available. Similar changes would be necessary to allow use of kernel-provided buffers that are mapped into multiple protection domains, obviating buffer copying (Druschel, 1994). An approach to providing nearly direct control of network adaptors on IBM RS/6000s in support of MPI collective communications routines is described in (Bruck, Dolev, Ho, Rosu, & Strong, 1994); the lower levels of their hierarchy would seem to fit well into our lowest level, with perhaps some small modifications to our mid-level. The extent of the changes necessary to support advanced network optimizations in the pC* communications hierarchy cannot be known until the attempt is made, but we are confident that the changes should be minimal.

Portability desires prevent us from doing the sort of vertical integration observed in (Turner, 1994), where data-parallel algorithms were coded in-kernel using a primitive message system, which was specifically adapted for the underlying FDDI interface, to implement some of the communications patterns described in this thesis. The characteristics of the FDDI network, the in-kernel implementation, and the limitation of testing to a fixed set of programs around which the system was designed permit very good performance, which pC* is only able to match on the largest problem sizes. The FDDI ring used in that research, though technically able to drop packets, proved to be sufficiently reliable that a "careful" protocol which assumed the network fiber would not drop packets was acceptable (Mosberger *et al.*, 1994); avoiding acknowledgement and retransmission support provided a significant performance improvement (roughly 20% of communication time). Although using a star network with EtherSwitch to connect our twelve machines resulted in a noticeable decrease in collision rate over a standard same-segment network, the synchronous communications behavior of data-parallel programs still results in periods where the network is overloaded and packets are dropped.

We have also examined closely the performance issues in attempting to impose a reliable protocol on top of UDP at user level, and the behavior of a variety of global reduction algorithms comparing the resulting system which supports multicast with a simpler point-to-point system. For our own current needs, the benefits of multicast are not sufficient to outweigh the costs of implementing a reliable system, and effort would be best directed to more general optimization techniques.

One such technique is a compiler analysis to permit communications routines to return before all messages have been received, leaving a key which can be used to block at the result's point-of-use until the communication has completed. The runtime infrastructure for this is almost complete, inherent in the message handling facility of the communications hierarchy which is needed anyway to deal with buffered messages in a timely manner. Only the analysis to detect the opportunities remains.

Version: comm.tex,v 1.14 1996/04/09 16:50:51 pab Exp

CHAPTER 5

ALGORITHMS FOR GENERAL COMMUNICATIONS

If a listener nods his head when you're explaining your program, wake him up.
— Alan J. Perlis, Epigram #17

Using the communications framework of the last chapter we describe an implementation of general communications—nodes send data to arbitrary nodes in arbitrary order. Our assumptions, for example that there is a fixed optimal message size, permit packing operations that would not otherwise be available without excessive data copying. We describe how a particular class of communications—those which send many values to the same address, or read the same address many times—can be detected at runtime and redundant data transmission avoided. The heuristic involved can determine that the overhead it is introducing outweighs its benefits, and turn itself off; when it is active, it can decrease runtimes by more than 50%. The heuristic can be tuned to a particular host platform and interconnect. We close with an experimental evaluation of the described system, comparing its common use with an implemented C extension similar to other optimizations which pre-compute communications schedules. We argue that evidence implies our more straightforward runtime-only approach is likely to be more effective, at least on our target applications.*

In the last chapter we presented the details of the pC* communications hierarchy, covering the functions that are available and the features that we can use in high-level operations to improve performance. We will now use these features to implement two of the simpler, yet most expensive, high-level communications operations: general send and get. We will start with a review of the semantics of these operations, then proceed to see how the features of the communications hierarchy are used to implement them. We then examine a special case in which the performance of the original implementation can be significantly improved. We close with a comparison with methods of handling the same problem in other data-parallel languages, where it is known in the literature as "irregular communication".

5.1 Semantics of General Communication

Recall that communications operations in C* are expressed through the use of left-indexing, a syntactic construct similar to array indexing in C but moved to the left side of the indexed expression to make it clear that the operation may involve communication with

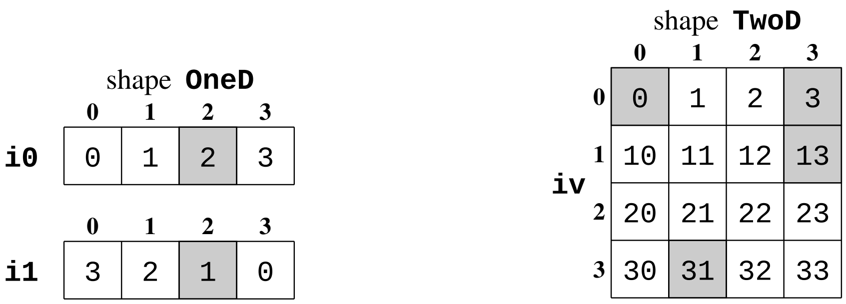


Figure 5.1: General Communications Operands

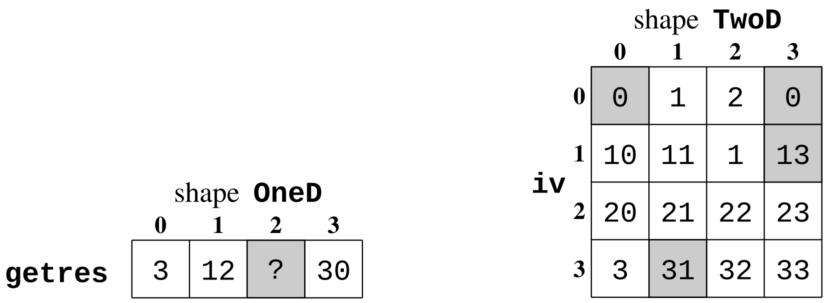


Figure 5.2: General Communications Results

other nodes. In the case of general communications, the index expressions are parallel integers of the current shape; however, the expression being indexed need not be of the current shape. There is an index expression for each axis of the indexed shape. Unless otherwise noted, throughout this chapter we restrict the notion of "left-indexed expression" to be specifically a left-indexed expression which results in a general communication, in contrast to scalar or grid left-indexing (cf. section 2.1.2).

When a left-indexed expression is used as a C *rvalue* or basic expression value, this is a "get" operation, and it results in a parallel value of current shape. The value at each local position consists of the scalar value from the indexed parallel expression at the "remote" position named by the global address specified by the local elements of the index expressions. For example, consider the system in figure 5.1. When evaluating the expression:

```
getres = [i0][i1]iv;
```

the current shape must be the one-dimensional 4-element shape `OneD`, so the index expressions are valid. Conceptually, each active position forms a global address using the index expressions and requests the corresponding element of the indexed parallel value. The resulting value of `getres` is shown in figure 5.2. Note that positions that are inactive in the indexing shape have no defined value, while positions that are inactive in the indexed shape will be read.

When used as a C *lvalue*, i.e. as the target of an assignment, a "send" operation is invoked. The results are similar, except in this case the index expressions name a remote position to which the value in the current position on the right-hand-side of the assignment will be sent. For example, we can modify certain positions in a parallel value with code similar to the following:

```
[i0][i1]iv = pcoord (0);
```

Here we use the index expressions to change the values in certain positions of `iv`. Again context is performed in the indexing shape, and because the third position of `OneD` is inactive the corresponding element of `iv` remains unmodified, as do any positions which were not named in the communication. However, the fact that position $\langle 0,3\rangle$ of `TwoD` is inactive does not prevent its value from being overwritten.

Note that in get operations we may read from the same position multiple times, and in a send write to the same position multiple times. Reading the same position results in no difficulties, since the value is replicated as many times as necessary. However, writing to the same position will result in “collisions”. C* permits these collisions to be resolved in different ways, using the compound assignment operators. If regular assignment is used and a collision occurs, exactly one of the values will be stored in the target position; the language does not specify which value will be stored. If compound assignment is used, the incoming values are combined with each other and the original value in the target position using the assignment's operator. This was exhibited in the idiom for image histogramming, described in section 2.1.2.

5.2 Basic Implementation Techniques

We have argued the value of walking through parallel data in linear fashion, to preserve good cache behavior (section 3.2.4). We would also prefer not to walk the shape more than once, for the same reason. Therefore the general communications implementations in pC* are based on a sequential walk through the current shape, converting index expressions to physical addresses in turn, and communicating with all nodes simultaneously. To avoid excessive memory use and improve communications latency we take full advantage of the features of the mid-level functions of the communications hierarchy.

The implementation of general send is straightforward. We maintain an MTU-sized buffer for each remote node. We proceed along the positions of the current shape in linear order. For each active position, we translate the global address from the index values into a node/offset pair. If the target node is our own node, we simply store the source value directly into its destination, performing combination as necessary. Care is taken to ensure the source and destination pvars are different, to avoid overwriting data before they are read. If the target node is remote, we add to the buffer for the remote node an integer naming the target position, and the value from the source pvar. When the buffer for a remote node fills, we transmit it and continue processing. Hence we are implicitly performing message vectorization (Hiranandani *et al.*, 1994) by combining values that are to be sent to the same node into larger groups to amortize transmission overhead, and achieve a degree of latency reduction by sending data as soon as they are available. We install a message handler which walks incoming buffers, storing the values into the locations specified by the source node; if a combining operator was specified, the operator is applied during the store.

The format of the send buffer is of some interest. To operate on data in or from the message directly (desirable for the optimization in section 5.3), alignment requirements for both offsets and data must be satisfied. An initially plausible method is to use a structure, containing one offset and one data element, to represent each packet, and placing these structures consecutively in the buffer. This is wasteful in several senses:

- Space is lost due to padding required between the end of the data element and the start of the next structure, aligned for offset access. In the case of sending character (one-byte) data, roughly $3/8$ of the message space is lost.
- The size of the structure must be determined dynamically, based on the size of the data element. This means that accessing elements of the buffer in turn requires a more complex address calculation (based on a run-time, rather than compile-time, stride).
- Larger data types, such as eight-byte `doubles`, may not have their alignment requirements met by implicit structure alignment. Addressing this requires either an additional modification of the inter-packet stride, or copying data out of the message into an aligned region with an expensive general copy routine.

Fortunately, because we know the length of the buffer, we can separate the offset information from the data blocks, and store the offsets in a contiguous aligned region at the start of the buffer, and the data in a contiguous region immediately following the offsets somewhere in the middle of the buffer. The start location for data can be determined by computing the maximum number of offset-plus-data elements that will fit in the buffer, and adjusting it for data alignment requirements. The number of elements actually present in the buffer is stored in the header of the send message. In the more common case where a buffer is completely filled no data moves are required. For the final buffer, where there is a potentially large amount of unused space between the last offset and the first data element, we can shift the data values down to the next aligned address following the offsets. This format ensures that both offset and data values are aligned in the buffer exactly as required for operations, and that space lost to satisfying the alignment requirements is minimized. Such an optimization is more complex when there is no bound on the length of a message; in that case we would have to maintain separate buffers for offsets and data, possibly combining them at the time the message is sent.

General get is similar, but two separate buffers are maintained for each node, since two communications are going on simultaneously: data requests and data responses. Again we walk the local shape computing node/offset pairs for each position. If the node is local, we read the value from the offset and store it into the result. If the node is remote, we add a packet to the request buffer, including the remote offset (where the data are to come from) and the local offset (where they are to be stored on reception). Again buffers are flushed as they fill. Two handlers are registered for a get operation. The first recognizes request messages and responds to them with the data requested and their target offset. The responses are in the same format as send data messages, so the second handler is the send handler described above: it reads in data from the remote node and stores them in the desired locations.

It is clear that some amount of bandwidth is wasted by sending the local destination offset to the remote node, to be returned along with the data. The alternative is to divine the destination offset from local information when the requested data arrive. This presents some complexity with respect to the possibility of receiving message buffers out-of-order (considered in section 4.2), and would at a minimum require either a temporary table proportional to the number of active positions on the receiving node, or re-walking the destination to detect active elements where incoming data should be stored. We prefer to leverage off the send implementation described above, and have not attempted a detailed examination to determine which implementation is better, leaving that for future work.[1]

Pseudo-code for the general get operation, including what each handler does, is given in figure 5.3. The `add*` routines pack their parameters into the proper format and add them to the buffer associated with the remote node, first sending off the current buffer if adding the new value would exceed the MTU. The `flush*` routines send off final packets to each node, marking each one as the last one for this communications operation. The handlers thus detect the completion of a given phase, allowing progress to the next phase: note that it is the handler for the request phase which flushes the responses (since no more requests will arrive). It is clear that after the initial walk of local data the node is doing nothing but waiting for messages to arrive and processing them inside the handlers. Here may be a good opportunity to take advantage of latency by moving the wait for completion of responses out of the communications routine back up into the user's code, right before the resulting value is referenced, as proposed in section 4.3.2.

The implementation here takes advantage of standard message-passing optimizations such as vectorization by combining information into MTU-sized buffers. There are rare cases where the size of a scalar element exceeds the network MTU; for example, when operating on a parallel value with structure elements. We detect the situation during the initialization phase, and allow the `addresp` and handler routines to operate on each element as a single message.

5.3 Optimized Send Operations

The implementation of general send described above places a message *packet*—a destination offset and a data value—into the buffer for each position in the shape; generally, somewhere between several dozen and several hundred packets fit into an MTU-sized buffer. This means that the number of network transmissions is proportional to the number of positions in the shape, scaled by the number of packets that fit into the MTU. Recall that

1. The enhancement described here was added subsequent to the completion of this dissertation. Though we cannot present an analysis in this footnote, initial results indicate that caching destination offsets locally cuts in half the execution time of get operations with no collisions on an Ethernet-linked cluster.

```
resp_handler (sender, msize, msg, parm) {
  for (i = 0; i < msg->npacks; i++) {
    parm->doop (&parm->target [msg->pack [i].offs], msg->pack[i].data);
  }
  if (msg->lastresp) {
    --nrespleft;
  }
}

req_handler (sender, msize, msg, parm) {
  for (i = 0; i < msg->nreqs; i++) {
    addresp (sender, msg->req[i].doffs, parm->spvar, msg->req[i].soffs);
  }
  if (msg->lastreq) {
    --nreqleft;
    if (0 == nreqleft) {
       flush_resp_packets (); /* Send off final response packets */
    }
  }
}

/* dpvar = [indices] spvar */
genget (spvar, indices, dpvar) {
  /* OMITTED: initialize parameters for resp and req handlers */
  nreqleft = nrespleft = MeshSize-1;
  register_handler (NextCode (MT_Send), resp_handler, &rpparm);
  register_handler (NextCode (MT_Get), req_handler, &rqparm);
  for (vp = 0; vp < numlocal; vp++) {
    (rnode, roffs) = convert_index (indices, vp);
    if (mynode == rnode) {
      dpvar [roffs] = spvar [vp];
    } else {
      addreq (rnode, roffs, vp);
    }
  }
  flush_req_packets (); /* Send off final request packets */
  while (0 < nrespleft) { /* Wait until we've received everything */
    if (0 > check_messages (io_timelimit)) {
      fatal ("timed out waiting for get.");
    }
  }
}
```

Figure 5.3: Pseudo-implementation of General Get

we have mentioned the possibility of collisions on both get and send operations, where multiple communications come from or are sent to the same position of a shape. When there are no collisions, the amount of network traffic is essentially optimal (modulo detailed analyses which obviate the need for some offset information), but in the case of collisions we waste bandwidth by performing collision resolution at the destination instead of the source.

To see this, consider again the image histogramming idiom of section 2.1.2. Under the above naïve implementation we require as many packets as there are positions in the source, `Image`, shape; generally on the order of one million. However, the destination shape normally has very few positions, say 256. If the addition combination for the histogram is done on the source node, we can drastically decrease the amount of communication required. We need only note that we already have a packet for a particular destination, and do the combination into the data region of that packet rather than creating a new packet. The trick becomes to recognize that a particular general send has this collision behavior, and take advantage of it. There are two issues here: the method used to detect that collisions are likely, and the method used to find the previously-buffered packet and form the combination.

Unfortunately, it is not easy to tell at the start of a general send whether it will or will not result in collisions. The case of histogramming, where the target shape is significantly smaller than the source, is one where we can be reasonably sure of collisions. However there are other high-collision cases where this does not occur—we may want to attribute to certain regions of an image (say elevation peaks) values from each pixel, sent to the nearest region center; for example, to determine the size of the surrounding area. In this case we have large numbers of collisions but the source and destination shapes are the same, and are often large. Similarly, experience with test and production programs showed that we cannot assume that sends with combination operators will result in collisions, nor that those with overwrite operations will be collision-free. Even a particular instance of general communication in a given source program may have strongly data-dependent collision behavior, causing compile-time analyses to fail. Therefore, the general solution requires either an implementation where collision detection is free, or one where we can turn the detection on or off within a particular communication operation depending on our success rate.

The simplest approach to collision detection is to scan through the buffer looking at packets already deposited, and stopping at a match. There is reason to believe that collisions are likely amongst close neighbors—in the case of histogram, adjacent pixels are likely to have nearly the same value. We can decrease detection time by first checking to see if we match the last value we stored, and if not scan from the end of the buffer backwards to slightly decrease the expected time to finding a match. From a theoretical viewpoint, the resulting linear scan through the buffer for each position is costly—$O(\mathrm{MTU})$—and we might want to use a more complex method with better asymptotic behavior, such as hashing on the offsets or binary search. Initially we chose not to implement these, because we felt a straightforward tight-loop scan through a relatively small contiguous area would be faster in practice than the alternatives, all of which require extra book-keeping that complicates both control flow and memory access. Experience from implementing the corresponding get optimization, described in section 5.4, led us to reconsider this assumption; we will ex-

Cluster Size	No Scan	Always Scan; Varying Collision Rates 0%	20%	40%	60%	80%	100%
4	3.80	5.36	4.68	4.11	3.46	2.88	0.61
8	2.16	2.80	2.56	2.19	1.93	1.64	0.33
12	1.57	2.03	1.71	1.61	1.38	1.17	0.24

Table 5.1: Send Communication Times: With and Without Collision Detection. Time in seconds to send 10^6 elements, clusters with 4, 8, or 12 nodes.

amine the issue in some detail later in this section. However, the collision detection method is not essential to the exposition here; the point is that any method will induce some amount of work, and that work is wasted when collisions are not occurring. It is the wasted work that we wish to avoid.

An understanding of the time taken in send operations, and in collision detection, can be taken from the performance numbers shown in table 5.1 and figure 5.4. The results are from tests performed on the star-network using each machine as a single worker, from 2 to 12 nodes. The test program used general send to store a constant 1 into certain elements of a one-million–element parallel `int`. The index pvar was carefully initialized so that stores were randomly distributed amongst the remote nodes (no local stores), and the collision rate within the send buffer—the likelihood the back-scan would find a match—was kept at a particular level. The program was run on different clusters with the backscan optimization both enabled and disabled, on index variables with collision rates from 0% (no collisions in a buffer) to 100% (all values went to the same position on each remote node). The depicted times where collision detection was enabled are the fastest runtime of five runs for each collision rate.[2] Testing indicated that, as expected, when collision detection was disabled, runtime is independent of the collision rate; the value given is the minimum of the times in five detection-disabled runs.

The cost of collision detection is observed by the difference in table 5.1 between the never-scan column and the always-scan column with a 0% collision rate. The experiment indicates that a 30–40% overhead is induced by scanning without success. This is a fairly high price to pay, especially in situations where we expect most send operations will have few if any collisions. However, the benefits of scanning are clear by comparing never-scan with always-scan on high collision rates. Scanning with a 100% collision rate is 85%—five times—faster than sending the values without scanning. Though the speedup here is an overestimate in general because the single index value was always found immediately, the histogram equalization algorithm with a 256-value pixel will also generally have a nearly 100% collision rate because the total number of positions per node is less than the number of packets that fit in a buffer. Tests on a histogram program indicate that the speed-up even with additional search time is over 65%.

2. Though we generally use the median as our statistic-of-choice for summarizing results, this particular test occasionally induced very bad network behavior, especially with low collision rates (i.e. high network traffic). The minimum was found to be a better filter of extreme results.

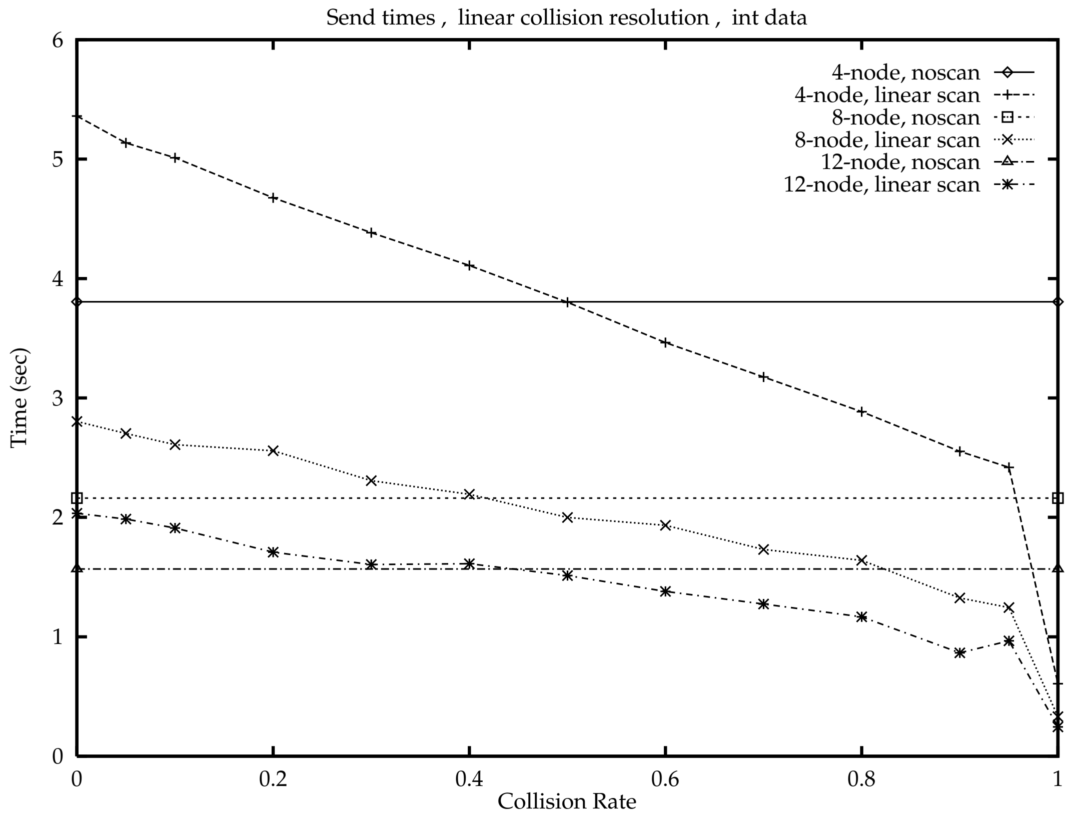


Figure 5.4: Send Communication Times relative to Collision Rates: Linear Scan

It is clear from these results that in some cases it is beneficial to search for collisions in send operations, and in some it is better not to do so. Since we are unable to determine collision behavior prior to beginning the communication, we wish to determine it as quickly as possible during the communication, so we can choose whether or not to perform backscanning.

Our solution to the problem is a heuristic which initially assumes there will be collisions, but keeps track of its success rate in searching for collisions. If the observed collision rate drops below a threshold value, the heuristic stops performing the search for a certain period of time, after which it restarts to see if the collision behavior has changed. In more detail, we use the following parameters:

- P_t is the threshold scan probability, which is the lowest observed collision rate at which it is still worth backscanning.
- B_I is the number of buffers that we scan while computing the collision rate.
- B_M is the maximum number of buffers that we will commit to scan based on past colli-

sion behavior. This ensures that if collision behavior varies over a single send operation we will not commit to scanning far into a low-collision region.

- B_S is the number of buffers on which we will not use detection if the collision rate drops below P_t. After this hiatus, we will recompute the collision rate.

The observed collision rate is estimated using an integer counter `collctr`, to which at each position a weight δ_s is added if a collision is detected, and a weight δ_f subtracted if no collision is detected. The weights are computed so that if the observed collision rate is P_t the counter will, on average, remain at its initial value, while if the observed rate exceeds P_t it will grow. I.e., we choose integer values for δ_s and δ_f such that:

$$P_t \approx \frac{\delta_f}{\delta_f + \delta_s}$$

The initial value of `collctr` gives us a window in which we can compute the observed collision rate without decrementing the counter to 0, which would cause us to stop scanning (hence stop observing the collision rate). To decrease the overhead of rate maintenance and dampen the effects of collision rate variance, we only examine the approximated rate when a buffer has filled. Hence, when a new buffer is started and scanning is turned on, we will continue to scan until the buffer fills, regardless of the behavior of the rate variable. The effect of this is that, if the rate counter is positive at the start of the buffer, we will scan throughout that buffer. Therefore, assuming that N_b packets will fit in an MTU-sized buffer, we set the initial counter to:

$$\texttt{collctr} = 1 + (B_I - 1) * N_b * \delta_f$$

This guarantees that, even if no collisions are observed, we will scan at least the first B_I buffers to compute our observed collision rate. Though this initial value biases the rate approximation, over time the effect is decreased.

There are two reasons for maintaining the maximum buffer value B_M. The first is that, when P_t is less than 0.5, $\delta_s > \delta_f$; and the closer P_t is to 0, the larger the difference. If we encounter a region of high collision, this means that `collctr` will be continually increasing. With values of P_t near 0, the magnitude of δ_s eventually may cause `collctr` to overflow, resulting in an apparent negative value and inappropriate disabling of detection. Secondly, if the collision rate varies over the shape, a period of high collision in this case may result in an extended period of detection even when the rate within the region is low, because many subtractions of δ_f are required to compensate for each past addition of δ_s. Therefore, we set an upper limit on `collctr` of

$$\texttt{collctr\_max} = 1 + (B_M - 1) * N_b * \delta_f$$

which ensures that, no matter how good past success was, if the collision rate drops to 0 we will scan no more than B_M buffers before disabling detection.

When the observed collision rate drops low enough that detection is disabled, we will stop the scanning step which is necessary to maintain the rate approximation. Since the rate may vary over a shape, we would like to re-sample the rate later on to see if it has changed. Therefore, when scanning is disabled we reset `collctr` to B_S, and decrement it each time the buffer fills. When it reaches zero, we reset it to its initial value and start scanning just as we did at the start of the communication.

Pseudo-code for the detection mechanism is given in figure 5.5. We maintain a separate detection flag `doscan` and collision rate approximation counter `collctr` for each remote node, because collision rates may differ depending on the destination of the send.

The final issue to be addressed is the choice of values for the parameters on which the heuristic depends. The appropriate value for P_t is dependent on a wide variety of system parameters, including processor speed (which affects scanning time), network transmission overhead and MTU, the size of the cluster, and the number of elements being sent. While an equation might be developed to calculate P_t for a particular configuration, perhaps at runtime, it is not immediately clear how all the system parameters interact (e.g. cache size versus buffer size), and it may be difficult to obtain the necessary system parameters. It is significantly simpler and intuitively more reliable to determine empirically the appropriate value from tests such as those depicted in figure 5.4. The cross-over points where scanning becomes beneficial are 50% for four nodes, 42% for eight, and 44% for twelve. We chose $P_t = 0.45$ as a value likely to perform reasonably well on all clusters. As long as an MTU-buffer holds a sufficient number of elements, say $N_b \geq 25$, we can get a good approximation of the collision rate within a single buffer, so we set $B_I = 1$. The remaining parameters are perhaps less critical, and we arbitrarily chose $B_M = 2$ and $B_S = 50$.

With these values, the performance is shown in figure 5.6. It is interesting to note that using the heuristic adds little cost when the collision rate is below the threshold, though a minute improvement could be gained by decreasing the threshold slightly for the eight and twelve node clusters. This is because the `collctr` value immediately drops below the threshold, and only one out of every B_S of the tens of thousands of message buffers created is scanned. As the actual collision rate increases, a few more buffers are scanned each time detection is re-enabled, resulting in a slight improvement due to collision resolution until the approximation again drops below the threshold. Finally, when the actual collision rate exceeds the threshold, scanning remains on throughout the communication and there is a significant drop in execution time.

The experimental results presented so far used a linear scan to determine whether a remote offset had been seen before. This seemed a reasonable choice, because the number of elements that fit in an MTU-sized buffer is relatively small: given a 1400 byte Ethernet packet (reduced by other message headers as described in chapter 4),[3] we can fit 172 4-byte packets and 275 1-byte packets, with offset information, in each buffer. This limitation does not hold for the get optimization described in the next section, because we wish to perform collision detection over the entire communication rather than within a single buffer. Since

3. See footnote 9 on page 90 for why sizes were limited to 1400 bytes.

```
if (Pt < 0.5) { /* Fixed value of 10 provides good approx */
  dfail = 10;
  dsucc = (int) (0.5 + dfail * (1.0 - Pt) / Pt);
} else {
  dsucc = 10;
  dfail = (int) (0.5 + Pt * dsucc / (1.0 - Pt));
}
initctr = 1 + (BI - 1) * Nb * dfail;
maxctr = 1 + (BM - 1) * Nb * dfail;
doscan = 1;
collctr = initctr;
reset buffer;
for each element in sequence do {
   if (doscan) {
      scan for element;
      if found {
         collctr += dsucc;
         /* perform any necessary combination in buffer */
         continue;
      }
      /* failed: adjust rate and fall through to store */
      collctr -= dfail;
   }
   /* Either not scanning, or failed to find element.  Flush an already
    * full buffer, since we're going to need another space. */
   if buffer is full {
      send off buffer;
      reset buffer;
      if (doscan) {  /* we were scanning---should we continue? */
         if (0 > collctr) { /* no, rate too low */
            if (collctr >= - Nb * dfail) { /* dropped below during walk */
              doscan = false;
              collctr = BS;
            } else /* neg because of overflow */
              collctr = maxctr;
         } else /* yes, truncate rate */
            collctr = MIN (collctr, maxctr);
      } else { /* we were skipping---should be restart scanning? */
         if (0 >= --collctr) { /* yes */
            collctr = initctr;
            doscan = 1;
         }
      }
   }
   add element to buffer;
}
```

Figure 5.5: Pseudo-code for scan collision detection heuristic

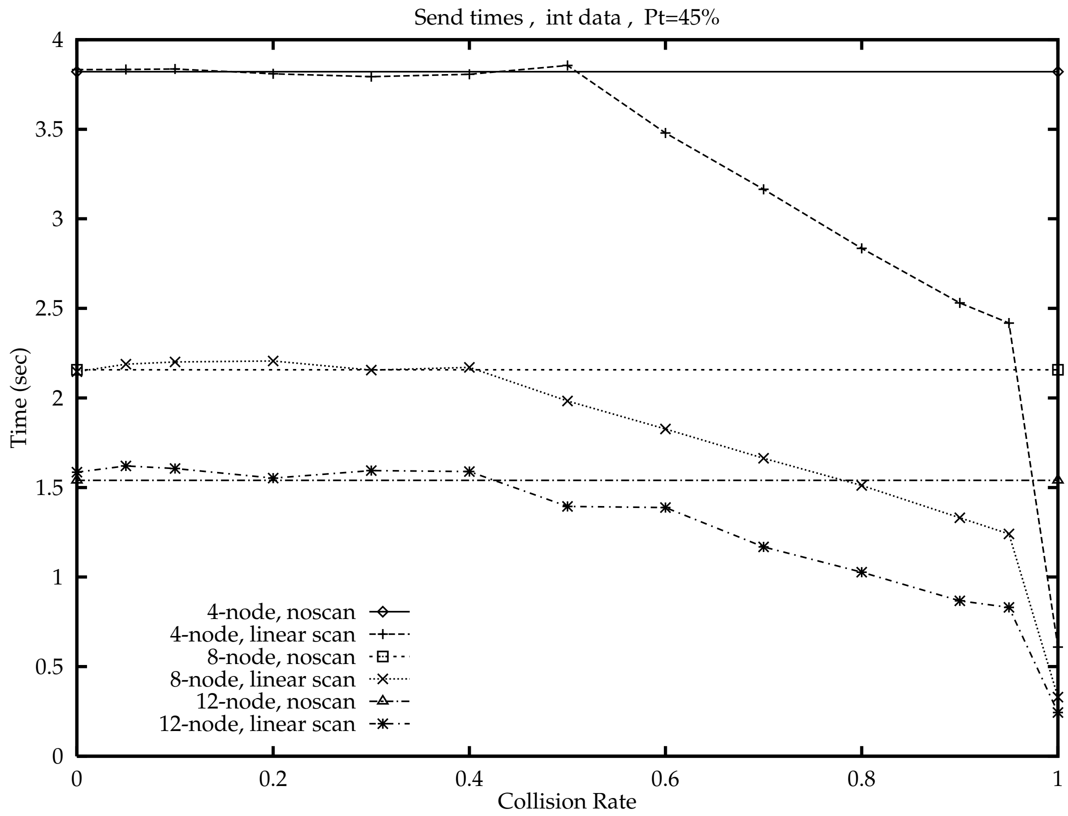


Figure 5.6: Send Communication Times relative to Collision Rates, with $P_t = 0.45$

the number of offsets we wish to track in that case is bounded only by the number of offsets on each remote node, we chose to implement an AVL search tree (Knuth, 1973) to improve collision search time. Imagine our surprise when we discovered that the cross-over point for an improvement using collision detection in the get implementation was essentially a collision rate of 0%: it cost us nothing to perform the collision search, even when it failed every time.

To resolve this anomaly, we re-examined the performance of a linear insertion compared with an AVL-based insertion for various numbers of elements, all with no collisions, and on our primary target hardware (Sun SparcStation 20/612). In each case, insertion required verifying that a particular key was not present in the search structure, then adding it to the structure. We verified that for small numbers a linear implementation is faster because the overhead is less: with 30 elements, linear insertion is twice as fast as AVL insertion (43µsec to linearly insert 30 elements, compared with 86µsec for AVL). However, the cross-over point is around 79 elements (275µsec each), and at 172 an AVL insertion is twice as fast (680µsec) as linear (1255µsec), and almost three times as fast with 275 elements. The net

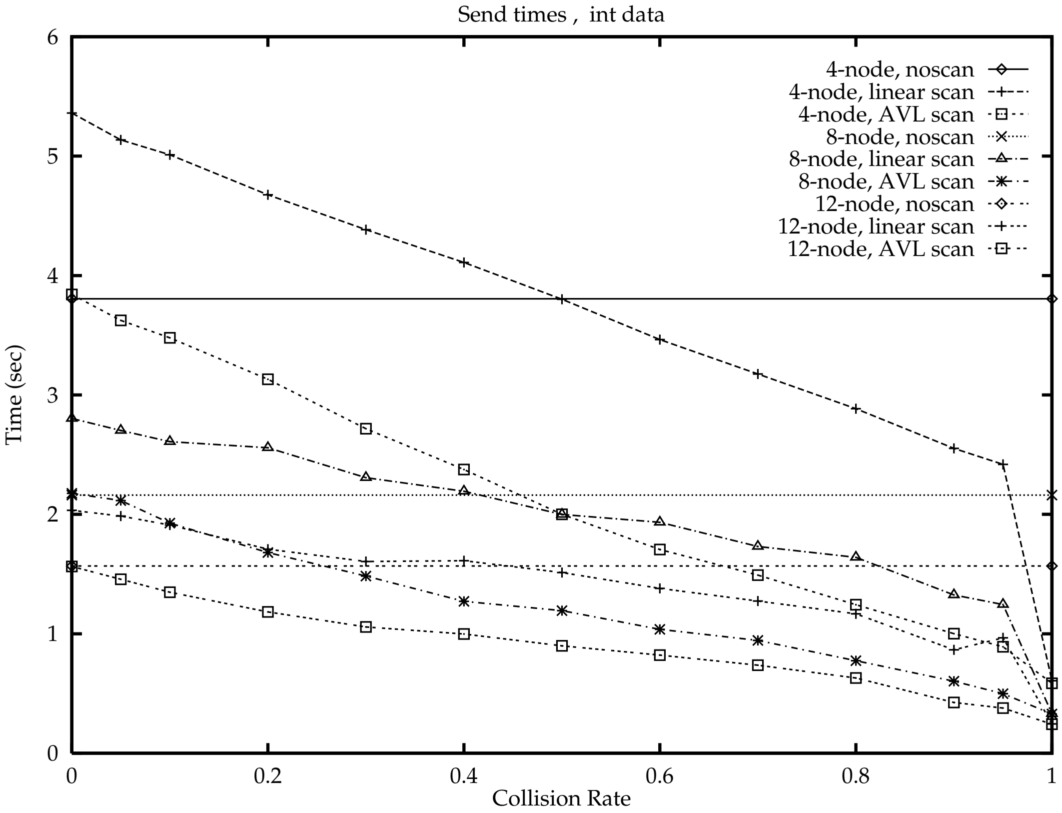


Figure 5.7: Send Communication Times, 4-byte Data, Linear and AVL Scan Methods. Buffers hold 172 elements.

---

effect of this is to reduce the overhead of search to the point where it is absorbed in the normal latency induced by the network and flow control, as further examined in section 5.5.[4]

We naturally modified the send implementation to use AVL trees to store collision information, and the resulting performance is shown in in figure 5.7 for integer data (172 elements per buffer). The 30–40% overhead observed for linear scan when no collisions were detected disappears with an AVL-based scan. It is satisfying to note that the real-time improvement using AVL instead of linear search is consistent across all but the very highest collision rates, making AVL twice as fast as linear at an 80% collision rate.

With character data (figure 5.8, 275 elements per buffer), the number of elements is large enough that not all AVL search overhead can be absorbed into normal latency, and the crossover point is somewhere around 20%. Comparing the two search methods, it is clear that

4. A similar effect is observed when a reasonable proportion of the data is sent to positions on the source node, thus bypassing the network entirely. This local work is also absorbed into communication latency, but reduces the proportion of time spent searching for collisions even more because no search is required to perform the combination/store to a locally-owned position.

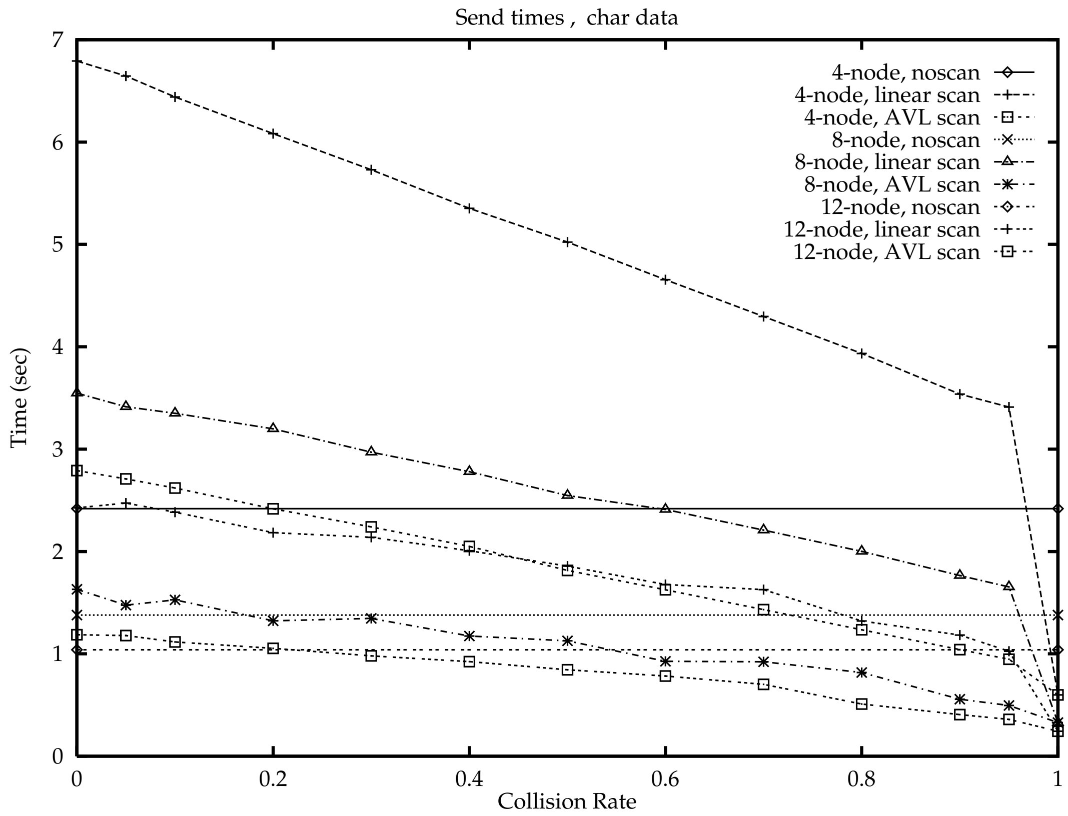


Figure 5.8: Send Communication Times, 1-byte Data, Linear and AVL Scan Methods. Buffers hold 275 elements.

our choice of $P_t = 0.45$ when using linear insertion is very bad for 1-byte sends on small clusters, and would yield a significant search penalty; a useful cross-over would be somewhere around 95% or higher.

The choice of an appropriate constant P_t for all send operations is less clear now that we have seen the effects on different sized data packets. Rather than further complicate the implementation by attempting to pick a rate at runtime, we now use AVL search consistently, with a threshold $P_t = 0.05$ to detect non-colliding sends.

5.4 Optimizing Get Operations

Collisions can occur in get operations as well as in send operations. In this case, we have a situation where multiple positions request the same (remote) value. Network traffic can be decreased if we recognize the case, request only one copy of the value, and when it arrives store it locally in the positions that need it.

The difficulty with optimizing get operations is that it requires saving state information, or walking the shape multiple times, to detect that a value requested some time previously and which has now arrived must be stored in several locations. To see this, recall that a general get is a two-phase operation. In the case of no collisions, it can be translated into two send operations at the C* source level. For example, with a rank-one shape, the get operation:

```
d = [idx]s;
```

can be written as:

```
[idx]ridx = pcoord (0);
[ridx]d = s;
```

The effect of this is to send our address to remote nodes which own the data we want, then have them send the data back. The complication with collisions arises because of the first send operation: if `idx` is not a simple permutation of the positions of the shape, then two or more address values from `pcoord (0)` will be sent to the same position of `ridx`, and only one will survive. The other is permanently lost, and the corresponding position of `d` will not receive the required value. We must retain this information.

To implement a get optimization similar to the previous section's send operation we must note, at the time the packet containing remote (source) offset and local (destination) offset is to be written into the request buffer, that a previous packet with the same source has already been requested. However, in this case we must perform the additional step of recording somewhere that when the requested data come in they should be stored in multiple locations. In the get implementation described in section 5.2, the store location is sent to the remote node to be returned along with the data, but there is only room for one store location in the send packet. Rather than attempt to expand the send packet to give multiple storage offsets, which would neutralize the benefits of not requesting the data multiple times, we must keep the multiple location information on the requesting node. When a packet comes in, we must determine whether it is to be stored in a single or in multiple locations, and what those locations are.

To support this, we maintain an AVL tree (Knuth, 1973) for each remote node. Each tree node contains as its key the source (remote) offset from which the value will be requested, and a list of local offsets into which the value should be stored. Since we have no bound (except the size of the shape) on the number of positions into which an incoming value must be stored, we arbitrarily choose a limit M_C (currently 6), and define a structure type which can hold M_C destination locations, plus a link to another object of the same type in case more than M_C destinations are required. As an optimization, we can detect when a series of gets are made from a single source position into a contiguous block of local elements, and store only the endpoints of the block. Thus we can handle arbitrary numbers of destinations, but in reasonably sized chunks to cut down on memory usage.

We use the detection heuristic of the previous section to determine whether the search is likely to be worthwhile, based on the observed collision rate. When collision detection is enabled and a new remote source / local destination pair is to be added to a request buffer, we first perform an AVL search to see if we already have collision data for the source position. If we do and the data have already been received, we simply copy the data over from a saved canonical location into the local destination. If we have a collision structure but the required data are not yet available, then we add the new local destination to the store list, and continue. If we have no collision information for the source offset, we scan through the current buffer of request packets looking for a collision. If we find one, we add a new entry to the collision tree and initialize its destination list to the destination from the packet in the buffer and the new local destination. We then replace the destination in the buffer packet with the value $-(s+1)$ where s is the source offset on the remote node. Legitimate offsets are always non-negative, and we will assume that no shape will have so many positions that a positive offset will appear to be negative due to overflow (i.e., for a 32 bit system we assume no more than 2 billion positions of a particular shape will be held on a single node).

When a data response arrives from a node, we walk each packet performing stores as before. However, if a destination offset in the packet is negative, we convert it back into a positive source offset and use its value to search the AVL tree for the current collision information for that offset. We then copy the value from the message buffer into all local elements where it is to be stored, and save a pointer to one of them as a "canonical source" in case additional requests for the value are made.

The performance of this optimization, relative to gets without collision searching, is shown in figure 5.9 for character data, and figure 5.10 for integer data. The benefits of collision search are apparent immediately, with any nonzero collision rate, and yield up to $8\times$ performance improvement with a 100% collision rate. As there is a very minimal overhead when no collisions are encountered, we set $P_t = 0.01$ for the get optimization.

The optimization here is a heuristic, and there are high-collision cases that it does not detect. An example is spread communications in two-dimensional shapes. Consider an $N \times N$ shape, where each position reads its value from column 0 of its own row. Because we perform a linear walk which proceeds along columns within rows, the first position requests the value from column 0; the second notices that the value has been requested and adds itself to the list; and so forth. In this case, each value is requested only once, and is then stored into all positions on the row.

If, on the other hand, each position reads its value from row 0 in its own column, then the linear walk means there are N intervening requests, all for different remote elements, before a redundant request is made. If N exceeds the number of elements that can be fit into the request buffer, that buffer will be flushed and a new one started before the next row is started, and the existence of collisions will not be detected.

Handling this case would require storing information about each request in the AVL tree immediately, rather than waiting until a duplicate is noticed. This would result in an AVL node for each remote request, yielding an extremely large search structure in the case of no collisions. Furthermore, since the distance between colliding elements may exceed the window in which collision rates are computed, we must either never disable search, or face the chance that we will turn off the search before we detect any collisions, thus wasting the

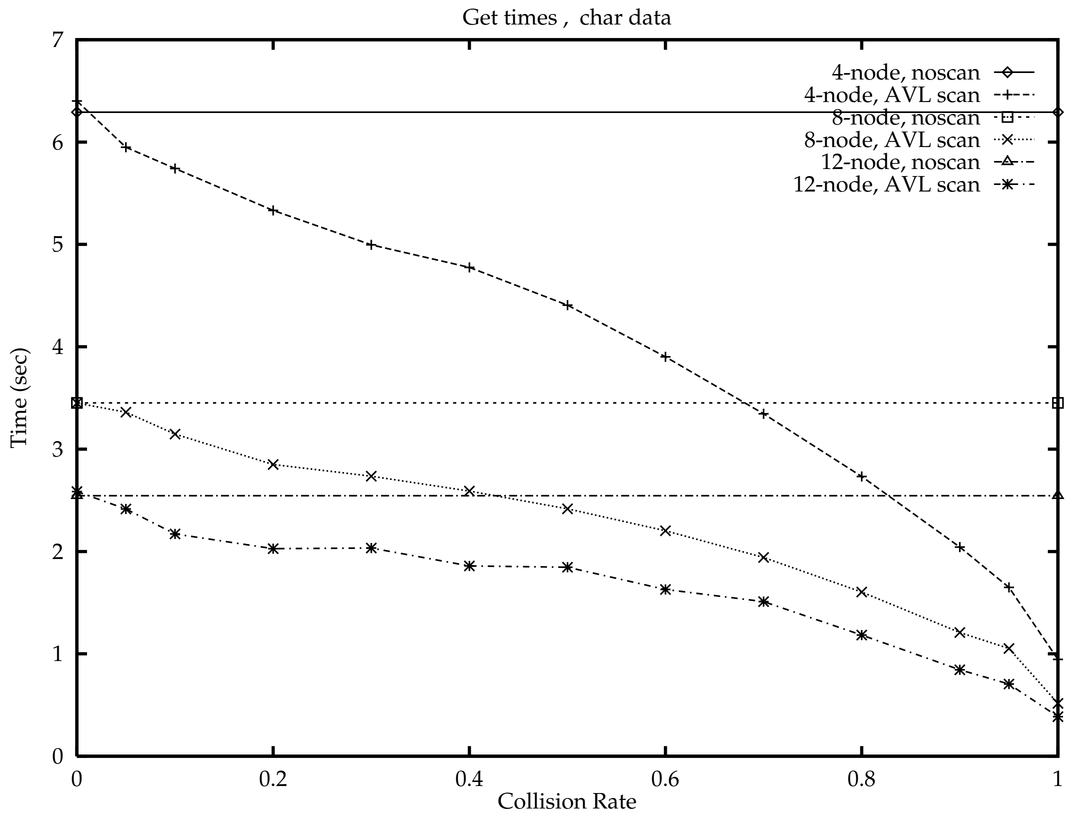


Figure 5.9: Get Communication Times, 1-byte Data

effort we spent building the search tree. Neither option is appealing, and we simply admit that there are colliding communications which are not detected by these heuristics.

5.5 Evaluation and Related Work

Implementation of the general case of what C* calls general communications has not been a major area of research, though at least one special case has received much attention. Most implementations use the same paradigm as Kali (Koelbel & Mehrotra, 1991) for these operations. The *inspector/executor* system inserts code prior to a loop which involves communications. The iterates of the loop are inspected in this code to determine the remote reference pattern and build a schedule that indicates what elements needed to be sent off-node, and where elements referenced locally could be found (either in the original local data, or in buffers holding off-node data). Then the loop itself is executed, using the schedule to speed up communication and indexing. This method has since been adopted and extended in current distributed Fortran implementations for a particular type of communication pattern, as

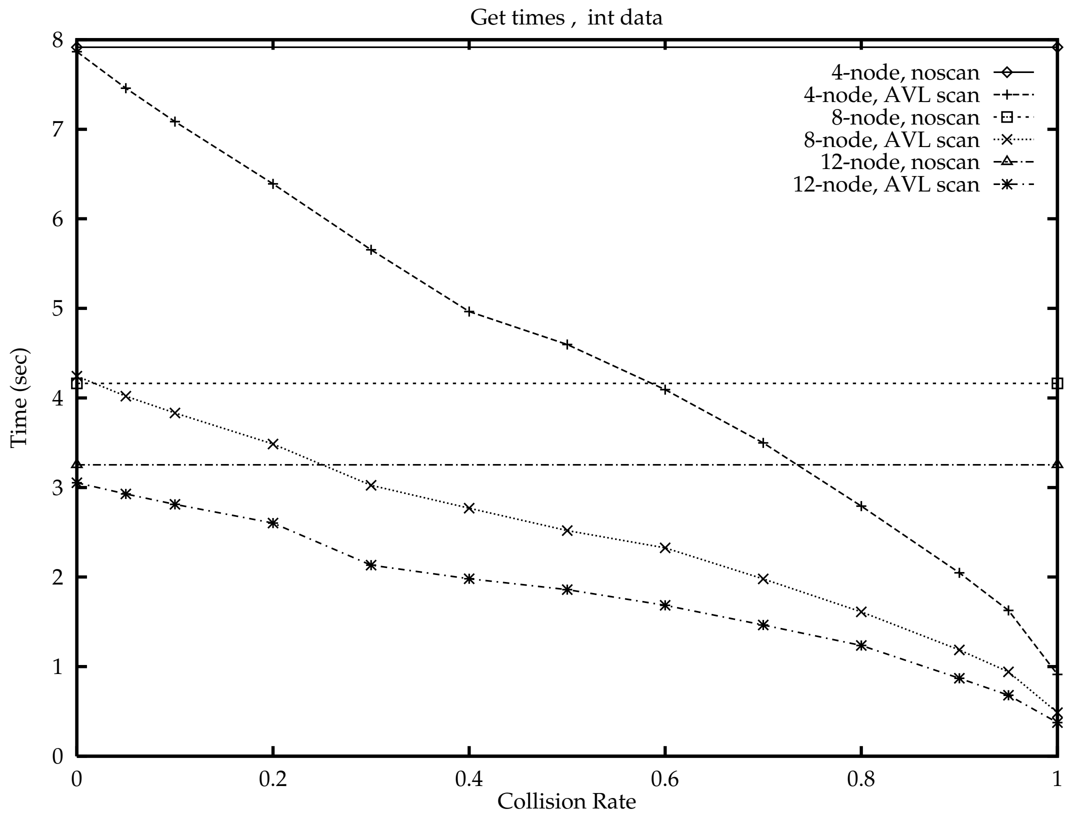


Figure 5.10: Get Communication Times, 4-byte Data

will be described below.

The University of New Hampshire C* compiler uses a more complex mechanism designed for a hypercube network architecture (Lapadula & Herold, 1994). Each node walks its local data and builds up a single large buffer that holds all data to be sent off-node. The buffers are then exchanged in a log-swap method, with receivers at each stage extracting the elements that are addressed to them, and adding new ones from previous steps. The need to buffer everything at once, and the multiple exchange steps which transmit values to nodes that are not interested in them, yield suboptimal performance on networked workstations. Recent versions of the UNH C* compiler support a variant of the schedule-based communication described below, memoizing the communications pattern the first time it is used and using point-to-point communication operations for further exchanges with the same pattern (Mason *et al.*, 1994). This implementation requires data-flow analysis to detect instances where the optimization is both valid and likely to be beneficial.

Several research programs in high performance distributed computing have addressed the issue of general communications in the context of "irregular problems" (Bozkus *et al.*, 1994; Chapman, Zima, & Mehrotra, 1994; Das, Uysal, Saltz, & Hwang, 1994; Ponnusamy, Saltz, Choudhary, Hwang, & Fox, 1995; Ponnusamy, Hwang, Das, Saltz, Choudhary, & Fox, 1995; Sharma, Ponnusamy, Moon, shin Hwang, Das, & Saltz, 1994; Brezany, Gerndt, Sipkova, & Zima, 1992). These are problems such as molecular dynamics and particle simulations where inter-element communication behavior is highly dependent on the data set, and is often adaptive, changing as the computation proceeds (e.g., as particles move about and affect different neighbors). The primary motivation for research into irregular problems is supporting *irregular distributions*, which ensure that the distribution of data is such that as little communication as possible is required between nodes: array elements are distributed based on the access patterns, as determined at runtime. For example,

```
DOALL i=1,ni
  DOALL j=1,nj
    A (i, j) = functionof (A (n1(i), n2(j)));
  END DO
END DO
```

where `n1` and `n2` are indirection arrays which map each array element to a different element (presumably, a "neighbor" in some sense), and `DOALL` indicates a data-parallel loop (which may ignore order dependencies). In C*, the computation would be expressed using general get and implicit data-parallelism:

```
A = functionof ([n1][n2]A);
```

It is generally the case that the indirection arrays involve references sufficiently far away that a block distribution would require inter-node communication for almost all references: there is no locality.

The most mature support for irregular distribution seems to be the Chaos runtime system developed at the University of Syracuse for the Syracuse Fortran-90D compiler (Ponnusamy *et al.*, 1995). The Chaos system supports irregular distributions at runtime through a six-phase sequence at each major computation loop:

- *Data partitioning* involves consulting a user-provided routine or directives to determine which processors should own particular elements of a distributed array. The result of this phase is a look-up table to map from array index to owning processor.

- *Data remapping* redistributes array elements in accordance with the distribution from the previous phase.

- *Iteration partitioning* determines which iterations of a parallel loop are to be executed on each processor.

- *Iteration remapping* redistributes the indirection arrays so that loop iterates own the indirection elements to which they will refer.

- *Inspector* examines the array indices for the node's iterates, and determines a communication schedule which indicates what values should be sent off-node, what values will arrive from off-node, and where off-node values will be stored locally.

- *Executor* performs the computation and communication using the results of the previous phases.

The lookup table of the data partitioning phase in essence defines a completely unconstrained data distribution (see section 3.1). Because there must be an entry in the table for each array element giving its owning processor and offset, the table must have as many elements as the array. In many cases, this requires that the table itself be distributed, lest memory requirements grow too large. Though Chaos attempts to limit the cost of accessing the distributed table by splitting it into pages which can be cached on nodes that normally do not own them, in the worst case determining the new owner of each position could involve a remote table lookup. Data mapping itself results in a general communication, sending data from their original owner to the processor appropriate for the upcoming computation loop.

Iteration partitioning is not directly relevant to C*, because loops over the scalar elements in parallel values are implicit, and are partitioned based on ownership of positions in a shape. Iteration remapping is intended to modify the owner-computes rule, by assigning responsibility for computing a value to the node which owns the bulk of the data involved in the computation. This has no analog in the pC* system, which uses owner-computes consistently, though other C* systems might differ.

The inspector/executor paradigm first introduced in Kali provides a measure of communication aggregation and latency removal, by collecting information that allows all sends to occur prior to loop execution, rather than performing a communication operation at each position in the array. There is support within Chaos and similar libraries to detect where a communication schedule can be re-used between iterations and in some cases between different code sections, and where values from off-node may be re-used without being requested again (Das, Ponnusamy, Saltz, & Mavriplis, 1992; Agrawal, Sussman, & Saltz, 1993; Ponnusamy, Saltz, & Choudhary, 1993). This re-use may be aided by compile-time analysis, or determined at runtime by destroying a schedule when an index expression on which it is modified.

The problem domain addressed by the Chaos library is quite different from that commonly encountered in image processing applications, and this is reflected in the very different communications implementations of Chaos and pC*. For example, the multiple phase preparation of Chaos can induce a significant overhead: in many reports of timing results using the inspector/executor paradigm, the inspector phase itself generally takes as long as a single executor phase (Agrawal *et al.*, 1993; Koelbel, Mehrotra, & Rosendale, 1990; Koelbel & Mehrotra, 1991; Ponnusamy *et al.*, 1993). When data partitioning and remapping is involved the preparation can be as long as 30–40 executions (Ponnusamy *et al.*, 1995). As a result, the cost of the preparation must be amortized over a large number of repetitions using the same communications pattern before an absolute improvement could be observed.

For the most part, a particular general communications pattern in a C* image processing program is used once, or perhaps a small number of times: for example, sending band data to a new shape as in example 3.2.3.1 need be done only as many times as there are bands in the multispectral image. If a C* program does use a pattern which is repeated multiple times, there is support within pC* to remove at least some of the overhead involved in determining the pattern at each communication. An extended data type, `CMC_sendaddr_t` (Thinking Machines Corporation, 1993), provides a representation of the address of a specific element in a specific shape; i.e. it encodes what would normally be an index vector as some internally-appropriate address. In the case of pC*, the `CMC_sendaddr_t` type holds the remote node and offset corresponding to a position in the shape. TMC C* and pC* both support library functions which translate a group of parallel index expressions into a parallel value which has a `CMC_sendaddr_t` at each position; there are also library `get` and `send` functions which use these address values instead of index expressions to perform general communication. This means that the address translation described in section 3.2.3 can be performed once prior to the communication, and the addresses used directly in communication operations.

This feature can be considered a crude approximation to a communications schedule-based approach. Experimentation in pC* indicates that the value of using this feature is highly dependent on the amount of communication involved and on network overheads. We devised a program to time several general send operations on a two-dimensional shape:

- Local stores—the indirection arrays named the local position as the target. No communication was involved.
- Shift—the indirection arrays shifted each node's data to the next node. For clusters with 2 or more nodes, all values were sent off node, but each element went to the same node as its predecessor.
- Cyclic distribution—the indirection arrays sent each position to a different node than the previous position. For clusters with P nodes, $(P-1)/P$ of the elements were sent off node, and there was no locality in sends.

Two send mechanisms were timed: the first performed the index-to-node/offset-pair translations on every communication, while the second first computed the `CMC_sendaddr_t` information for the communication into a parallel variable, then used the communication operation which read the pre-computed node/offset data from the parallel value.

To see the performance effects of pre-computing the node/offset information, we measured the time to perform each of the sends with each implementation from one to five times in sequence, on a 1024×1024 shape with one byte per position. The programs were run on clusters with 1, 2, 4, and 8 nodes. For each cluster size and communications pattern, we fitted the data to an estimator function appropriate for the send model: normal left-index sends with address recomputation cost a particular amount for each repetition, while `CMC_sendaddr_t` sends have a fixed overhead plus a different amount for each repetition. The estimator functions, yielding runtimes in seconds, are given in table 5.2.[5]

5. Tests were performed on the compute cluster using the reliable TCP interface, with one process per physical machine. Times for per-iteration costs were found by taking the median of five runtimes for each of 1,

Cluster Size	Send Pattern	Recompute Send	`CMC_sendaddr_t` Send	Per-Iter %Imp Using SA	Min Iters to SA Faster
1	local	$2.34n$	$2.95+0.99n$	58	3
	shift	$2.35n$	$2.95+0.99n$	58	3
	cyclic	$2.39n$	$2.95+1.04n$	56	3
2	local	$1.44n$	$1.75+0.49n$	66	2
	shift	$4.75n$	$1.74+4.59n$	3	11
	cyclic	$2.52n$	$1.72+2.30n$	9	8
4	local	$0.78n$	$0.92+0.25n$	68	2
	shift	$2.57n$	$0.93+2.49n$	3	13
	cyclic	$1.99n$	$0.88+1.88n$	5	9
8	local	$0.45n$	$0.53+0.13n$	71	2
	shift	$1.29n$	$0.53+1.26n$	2	17
	cyclic	$1.28n$	$0.49+1.25n$	3	14

Table 5.2: Time estimators for n repetitions of various send communications (sec)

In all cases, the precomputation itself is more expensive than computing the address in the context of the send operation. This is mostly due to the need to allocate and fill a parallel variable containing node numbers and offsets for each element in the shape—the analog of a communications schedule—and may be a cause of the similar overhead in inspector/executor implementations. When we consider the total cost of communication, things become more murky. With one processor—forcing all moves to be local—address pre-computation is beneficial if three or more sends are to be performed; the results indicate that about 60% of the per-iteration cost is due to address calculation. With multiple nodes, where sends actually invoke communication costs, only the send pattern with no off-node transmission yields an impressive improvement with pre-computation. In the other patterns, the cost per repetition is very nearly the same with each method, with about 3% savings with address pre-computation. It would require between 8 and 17 iterations before the overhead incurred by precomputation was recaptured. This is surprising, considering the result in chapter 3 which implied that address calculation would be 10% of the communication cost; only one communications pattern / cluster size supports an overhead that high.

The conflict can be resolved by recalling that the previous estimate of 1msec for sending a 1400-byte Ethernet packet did not take into account overhead from flow control. More than 100KB of data are transferred off-node in each iteration of shift and cyclic patterns in these tests. To prevent deadlock due to filled the system network buffers, the `check_messages` routine will be invoked to read and manage incoming data while the out-

2, 3, 4, and 5 repetitions of the communication. Four per-iteration costs were found by subtracting pairwise (5-4,4-3,3-2,2-1). The mean of these costs is the value given in the table. The variance of the four costs was generally around 1% of the mean, with a maximum of 8% for the 8-node cluster, indicating that the approximations are a close fit to the data independent of number-of-iterations. The time for the send-address build is the mean of the measured time for the build, rather than an extracted intercept from fitting the recorded times to a line.

going data are still being prepared and sent. While these invocations are critical for correctness, the additional overhead increases the per-buffer cost to between 2.4 and 3.2 msec for the communication times in table 5.2. Tests indicate that the same behavior holds for smaller shapes as well, because latency is small enough that incoming messages arrive before all local messages can be sent off-node. Though the underlying network would be capable of buffering the entire communication, the system cannot know this, and transmission is delayed to handle the incoming messages. The larger transfers are also slowed due to operating system-induced flow control (e.g., writers must wait until a reader has acknowledged receipt of previous data before more data can be queued).

While it is still beneficial to perform address pre-computation when most communication is intra-node, the inherent costs of communication, both in overhead and in latency, make pre-computation less valuable for heavy communication patterns. To some extent, this is due to the choices made in pC*'s communications hierarchy: for example, avoiding unportable, specialized network-specific optimizations that could reduce the overhead, and using fixed-sized buffers rather than sending all data in one buffer. However, the results do imply that the communications schedule approach may have certain limitations which make it less appropriate for systems where communications operations are removed to library routines, not specialized to a particular source-program loop. Few papers have compared communication times using a schedule implementation with a well-optimized implementation which computes addresses at each position. The small fractional improvement we observe with our approximation to schedule-based communication implies that other systems may see similarly little improvement when compared with non-schedule implementations which do not have the overhead of the inspector loop, especially for patterns with few repetitions.

The University of New Hampshire C* system supports an optimization which builds a communication schedule during the first execution of a communication pattern, and uses it for subsequent repetitions (Mason *et al.*, 1994). This optimization yields speedups of 10% or more per repetition on networked workstations when the schedule approach is used, but this seems mostly because the unoptimized approach uses a log-based communication which passes values through intermediate nodes on the way to the final destination. Both implementations are generally two to three times slower than the corresponding code in pC* running with the same network interface (PVM) (cf. section 7.8). The experimental results in (Mason *et al.*, 1994), which use the Intel Delta architecture for which the non-schedule implementation in UNH-C* was designed, do not clearly quantify the expected benefit of the optimized implementation.

One clear benefit of the communication schedule approach of Chaos and Kali is its potential to access local elements directly, by providing a schedule which points to local data in their local positions and remote data in a separate buffer. Buffers for remote data in communication schedule systems are often allocated as "ghost cells" at the end of the local portion of a parallel value (Das *et al.*, 1992). In the worst case of accessing only remote data, the external buffers must be the same size as the local data, meaning that no space savings is achieved; in fact, more space is required to store the schedule itself. However, the copying cost will be avoided for intra-node references. This is in contrast to pC*, which uses a temporary parallel value into which both local and remote values are copied, to place values at the element at which they will be referenced. This simplifies the implementation considerably, at a cost in extra memory and copying.

The performance results in (Ponnusamy *et al.*, 1995) indicate that the bulk of improvement using the Chaos runtime system comes directly from its support for irregular distribution: the system ran twice as fast in its executor phase when data was re-distributed to limit inter-node references, although the preceding five phases took somewhat longer to perform the more complex preparation. The partitioning scheme is algorithm and data dependent, and is not a direct part of the Chaos system. If irregular algorithms with repeated execution are to be coded in C*, the same functionality can be achieved within the core C* language without additional compiler support, assuming a repartitioner is available. This is done by noting that data redistribution is essentially a renumbering of the distributed elements, so that communications nearness is reflected in their ordering.

For example, let `d` denote a (rank-one) parallel value, `i` an index vector, and `m` a mapping vector taking position `k` to position `[k]m`. That is, `m` encodes a partitioning (a permutation of element addresses) designed to decrease inter-node communications while performing communications such as `[i]d`. Then the parallel value `d` may be redistributed with a general send:

```
[m]rd = d;
```

and index expressions remapped through a general get:

```
ri = [i]m;
```

After this redistribution, get operations of the form `[ri]rd` yield the same parallel value as the original `[i]d`, though with less inter-node communication (assuming `m` represents a better distribution). Redistribution of higher-rank data can be supported in a similar fashion. This technique is used in the Julia-set benchmark given in section B.6, to impose a cyclic distribution of rows so the computational load is balanced.

These idioms show how any arbitrary distribution may be supported in pC* even though the underlying system supports only block-based distribution. The caveat is that the user must undertake to specify the required distribution and perform the redistribution steps herself. However, this is not much different from HPF and Fortran-D which similarly require source-level hints or calls to extrinsic repartitioners to provide the necessary distribution information (Chapman *et al.*, 1994; Bozkus *et al.*, 1993; Ponnusamy *et al.*, 1995).

5.6 Conclusions

Two of the core communications operations of C* involve sending and getting values from positions which are related in arbitrary ways to the target position. In this chapter, we have presented a straightforward yet effective implementation of these operations using the communications hierarchy described in chapter 4. We have also presented a heuristic which, at essentially no cost, detects at runtime a special case where the amount of data sent over the network can be reduced, resulting in a significant performance improvement.

The contents of this chapter are only distantly related to other work on similar communications patterns, which are focused on pre-computing send and receive behaviors to aggregate messages and take advantage of latency. These methods tend to involve sufficient overhead that for the common (in image processing) case of few executions of each pattern we feel the straightforward, one-pass implementation is preferred. We achieve message aggregation through the use of MTU-sized buffers, and take advantage of latency by sending buffers as soon as they are filled, or (in the future) by delaying the barrier that verifies all data has been received until the resulting value is to be accessed.

While the irregular communications patterns addressed in this chapter have inherent limitations on runtime optimization simply because they can distribute data arbitrarily, patterns with regular behavior, such as shifting data along a vector, admit a highly optimized implementation which can be up to ten times faster than a general communications operation with the same communications behavior. These optimized grid communications are the subject of the next chapter.

CHAPTER 6

GRID COMMUNICATION

You can't communicate complexity, only an awareness of it.
— Alan J. Perlis, Epigram #105

Here we develop an optimized method of calculating the communications requirements for grid-based sends and gets completely at runtime, in contrast to most other systems which require that knowledge of shape (array) size and runtime environment be available when code is generated. The mechanism works on shapes of any dimension or size, shifts in any number of dimensions simultaneously, clusters of any number, and any data layout conforming to the generalized block distribution described in chapter 3. It permits a common contextualization operation to be performed roughly $100\times$ *faster than a method which looks at each position. Experimentation on grid communication indicates it imposes no more than a 5% increase over the cost of simply copying data, up to the point network latency interferes. Experimentation on a single processor indicates its performance is nearly independent of shape rank, is competitive with hand-coded optimized implementations for one- and two-dimensional send operations, and is orders of magnitude faster than a general method which performs translations at each position in the shape.*

Many algorithms in both image processing and scientific computation use neighborhood or stencil operations where elements near a position are combined to create a new value for that position. These operations use grid communications: elements get values from positions that are a fixed offset from them through one or more dimensions. In the case where array bounds, offsets, distribution, and number of processors are all known at compile time, determining what values are required can be done by the compiler, which can then emit explicit calls to message passing routines to send data to the nodes which will want them, read data from other nodes, and perform local computation.

With C*, none of this information, including potentially the rank of the array being indexed, may be known until runtime. We must still provide a way to read values from fixed offsets with as little overhead as possible. In this chapter we will describe a method of global-to-local address computation for grid-based communication which is completely resolved at runtime, up to and including the rank of the shapes it is to work on. The implementation is more than competitive with previous runtime resolution schemes, and provides ability to overlap communication and computation both within the library routines which accomplish the communication and, with the addition of some compiler support, across the user's code as well.

Because of the intricacies of a fully-general resolution scheme handling arbitrary distributions (of the sorts described in section 3.1), we build up to the grid algorithm by first examining the simpler but related case of forming the run-length encoded context map for boundary-restricted contexts, which uses the same fundamental concepts.

6.1 Forming Grid Boundary Contexts

In most of this dissertation, we have viewed the local component of a distributed parallel value as a linear sequence, abstracting away from whatever multi-dimensional system the user has imposed on it. This allows us to handle arbitrarily ranked shapes without concern, but causes difficulties when working with operations that inherently depend on the users' view of the data, such as grid offset computations.

If the shape's rank is available to the compiler, an alternative to the linear VP sequence is a set of nested `for` loops, one in each dimension, ranging from 0 to the extent of the shape in that dimension. In the case of distributed data, the bounds for a particular node would consist of the subgrid that the node owns. This is the approach taken by most compilers for distributed systems (Tseng, 1993), and makes computations which rely on the user's view of the data fairly easy to implement, since the global positions along particular axes are immediately available through simple transformations of the index variables. However, the approach generally relies on having not only rank but also extent, grid offset, data distribution, and the number of processors in the target system available for the compiler.

As noted previously, C* does not guarantee us knowledge of shape rank at compile time, and our implementation is required to work with code where shape dimension and cluster size are not specified until runtime. To support run-time specification of shape rank, not all communication in C* uses the left-indexing syntax introduced in section 2.1. There are grid communication library routines which handle arbitrarily-ranked shapes, with the offset vector specified either through C's `stdarg` support for variadic functions, or though an array of integers. If the system implemented one version for each rank of shape that might be handed to the routines, it is clear that unreasonable code bloat would occur and only a small number of ranks could be supported. Therefore, we need a way of simply but effectively emulating a multidimensional nested `for` loop using a single-nested iteration construct. Code to perform this operation is shown in figure 6.1, and forms the foundation for all the methods used in this chapter. Briefly, we use an integer array `idx[]` to represent the loop indices, and increment them from highest axis (deepest loop) to lowest axis (outermost loop) in turn, wrapping when each reaches its upper bound, until the outermost loop wraps, at which point we terminate. Since data are laid out in row major order, we encounter each position in turn in its linear order, preserving good cache behavior. As a lagniappe, the code automatically resets all index values so it is ready to execute again. Let's now see how this approach can be used to generate the run-length encoded context for a boundary-restricted context.

Boundary contexts were described in section 3.3 as blocking off regions of a shape certain distances from the boundaries of each axis, to prevent out-of-bounds access through grid operations or preserve border information in iterative algorithms. The type of restric-

```
/* Assume lb and ub are initialized to the lower and upper bounds,
 * depth to the number of dimensions, and idx[k] to lb[k].  Generates
 * cross-product from [lb_k,ub_k). */
do {
  /* OMITTED: Perform op for idx[0],idx[1],...,idx[depth-1] here. */
  k = depth;
  while ((0 <= --k) && (++idx [k] == ub [k])) {
    idx [k] = lb [k];
  }
} while (0 <= k);
```

Figure 6.1: Emulation of Arbitrarily Nested `for` Loops

tion recognized by the pC* compiler is a conjunction of one or more comparisons between a `pcoord` call and a scalar integer expression. As an example, consider again the context which prevents undefined behavior when executing the communication of figure 2.2:

```
where ((dimof(current,0)-1 > pcoord (0)) &&
       (0 < pcoord (1))) {
  iv2 = [.+1][.-1]iv;
}
```

The bounds for the active region use the same values as distribution partitions use for node boundaries: the lower bound names the first active position along the axis, while the upper bound names the first following inactive position, so the desired range is $[\mathtt{lb}, \mathtt{ub})$. The first conjunct in the example lowers the global upper bound on active sequences along axis 0 from `dimof(current,0)` by one; the second raises the global lower bound on axis 1 to 1.

The pC* compiler will recognize `where` restrictions of this form, and translate them to calls to a library routine which is given the current shape, parent context, and a sequence of triples naming axis, comparison operator, and integer bound value. Each node then initializes the bounds of its subgrid using the boundaries for the portion of each axis that it holds. Where a restriction is given that blocks off part of the region held on this node, we move the corresponding bound inwards to note where the active region starts or ends.

Conceptually we then have the following information for each axis k: the lower bound of data held on this node d^l_k, the upper bound of data held d^u_k, the lower bound of the active region a^l_k, and the upper bound of the active region a^u_k. Since we are interested only in the active region on this node, we can ensure that $d^l_k \le a^l_k$ and $a^u_k \le d^u_k$. Therefore, we could form the context by walking the loop using the $\langle d^l, d^u \rangle$ pairs as bounds, and at each position checking to see whether `idx` falls within the $\langle a^l, a^u \rangle$ bounds. However, we can do much better than this.

First, note that the active sequence on the highest axis is a contiguous block from a^l_{r-1} up to a^u_{r-1}. Therefore, we never need to increment through the highest axis point-by-point: for each index set of lower axes, we get an active sequence of size $(a^u_{r-1} - a^l_{r-1})$ positions. Furthermore, if it should happen that $a^l_{r-1} = d^l_{r-1}$ and $a^u_{r-1} = d^u_{r-1}$, then the

Name	mulfact	noob	nb	Loop 0			Loop 1			Context
				lb	ub	oob	lb	ub	oob	
unrestr	24	0	0	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	*n/a*	24a
bottom	6	0	1	0	3	6	*n/a*	*n/a*	*n/a*	18a 6i
lr	1	1	2	0	4	0	1	5	2	1i 4a 2i 4a 2i 4a 2i 4a 1i
rtb	1	6	2	1	3	12	0	5	1	6i 5a 1i 5a 7i

Table 6.1: Context Build Info for Example Boundary Restrictions

same contiguity argument propagates down to axis $r-2$, giving active sequences of size $(a^u_{r-2} - a^l_{r-2})(d^u_{r-1} - d^l_{r-1})$, and we do not have to walk the highest axis at all. By similar reasoning, unrestricted internal loops can be merged and replaced with a new loop which iterates up to the product of the ranges of the unrestricted loops.

The second thing to note is that the length of inactive sequences is also known immediately. If $d^l_{r-1} < a^l_{r-1}$, then we start with an inactive sequence of length $(a^l_{r-1} - d^l_{r-1})$. When we wrap around an axis k, we are skipping an inactive sequence of length

$$((d^u_k - a^u_k) + (a^l_k - d^l_k)) \times \prod_{m>k}^{m<r} (d^u_m - d^l_m)$$

which skips the lower and upper inactive areas and scales by the number of positions on higher axes which are disabled by out-of-bounds indices on axis k. Taken together, these observations mean that we can restrict the bounds of the iteration space to those of the active region, with axes that have no limitations merged together, and detect inactive regions in chunks when we wrap an axis in the loop emulation. The code in figure 6.2 builds the necessary data. At the end of this code, `nb` indicates the depth of nesting that we must use to generate the context. `noob` contains the length of the initial out-of-bounds (inactive) sequence; all other inactive sequences arise from wrapping around axes during the iteration emulation, and a wrap on axis `k` yields an inactive sequence of length `oob[k]`. `mulfact` gives a scaling factor representing unrestricted axes at the high end of the iteration space.

Several example contexts on 4×6 grids are shown in figure 6.3, with the corresponding loop and context information in table 6.1. Note that if there are no restrictions, `nb` $= 0$, and we can immediately store an encoding for an everywhere-active context. The corresponding everywhere-inactive sequence can be detected by finding a case where `lb[k]` $=$ `ub[k]`.

We can now generate the run-length encoded context using the template in figure 6.4, assuming the parent context is everywhere-active. The omitted code is essentially that required to store an RLE sequence, as shown in section 3.3. In the presence of a parent context, the code is somewhat but not informatively complicated by the need to merge inactive/active sequences with the restriction imposed by the external context.

The time taken by the context forming loop is proportional to the number of context sequences in the resulting RLE encoding, with a slight overhead to wrap the loop counters.

```
mulfact = 1; /* Scaling factor for unrestricted axes */
nb = 0; /* Number of nested loops to simulate */
k = 0; /* Axis number being examined */
pprod = ShpNumLocal (current); /* Number of pos per index at axis */
noob = 0; /* Number of positions out-of-bounds at start */
while (k < rankof (current)) {
  pprod /= ShpDimLocal (current, k);
  if ((al [k] == dl [k]) && (au [k] == du [k])) {
    /* Combine unrestricted axis with adjacent unrestricted axes */
    mulfact *= du [k] - dl [k];
  } else {
    if (1 < mulfact) {
      /* Store a loop to cover combined unrestricted axes */
      idx [nb] = lb [nb] = 0;
      ub [nb] = mulfact;
      oob [nb] = 0;
      nb++;
      mulfact = 1;
    }
    /* Set the bounds */
    idx [nb] = lb [nb] = al [k];
    ub [nb] = au [k];
    /* Compute leading out-of-bounds region */
    oob [nb] = (lb [nb] - dl [k]) * pprod;
    noob += oob [nb];
    /* Compute trailing out-of-bounds length */
    oob [nb] += (du [k] - ub [nb]) * pprod;
    nb++;
  }
  k++;
}
```

Figure 6.2: Build Procedure for Boundary Contexts

For a two-dimensional shape with $N \times M$ positions, the number of context sequences will be at most $O(N)$ with a restriction on axis 1, and a constant at most 3 with an unrestricted axis 1, in essence eliminating the factor of M from the context formation time. As noted in section 3.3, the ability to avoid evaluating multiple `pcoord` expressions at each position in the shape can provide a significant improvement in speed.

6.2 Application to Grid Communications

As with general communications, grid communications come in two flavors—send and get—and an overview of their properties and implementation is in order. In both cases, communication involves moving elements along a constant vector in one or more axes; the vector is specified by an array of signed integers, with as many elements as the rank of the

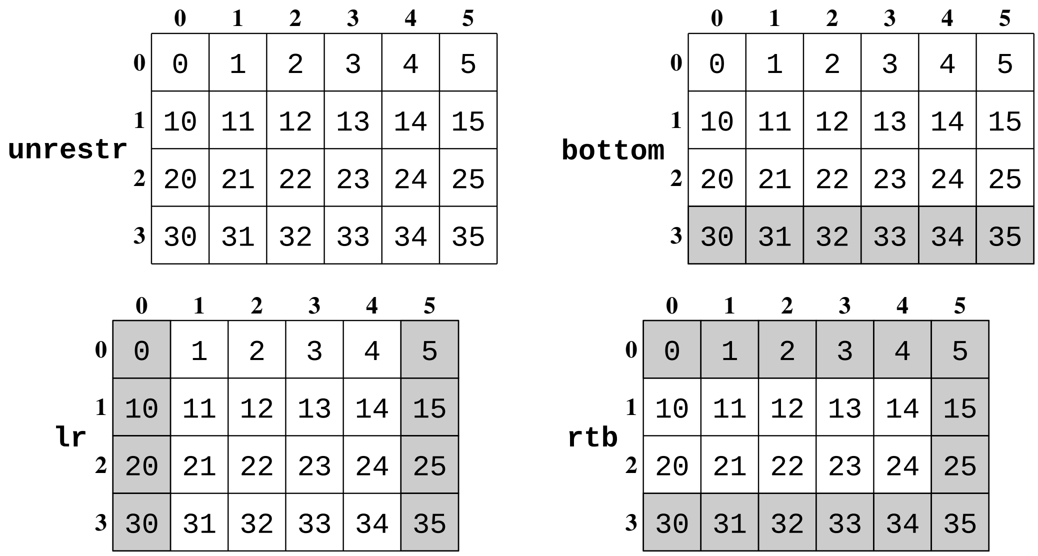


Figure 6.3: Example Boundary Restrictions

communicating shape.

In a grid send operation with offset vector $\vec{v}$, each active position at global location $\vec{p}$ sends its value to the position $\vec{p}+\vec{v}$. The position receiving the value may either combine it with the current value at that position or replace its value with the new one, just as general send may combine or replace. If the target position is out-of-bounds, the value is not sent. However, some interfaces to the basic grid operation permit a fill value to be specified: when the *source* location is out-of-bounds (i.e., an element has nothing sent to it), the target instead reads a value from its position in the fill value. This can be used to define boundary values which are used by default when communications extend beyond the area we are primarily interested in (and for which we defined the shape). Grid get operations are slightly different, in that each active position $\vec{p}$ requests the value from position $\vec{p}+\vec{v}$. If the latter position is out of bounds, a fill value may be read instead. In both cases, if the grid operation does not provide a fill value, we would like the option of having the system detect where out-of-bounds positions are accessed, since this may result in undefined behavior on other systems or future versions of the pC* system.

At a high level, both communications require the same operations: we walk through the positions on this node, sending data or requests to other nodes, and using the incoming data or fill values. Naïvely, for each local position we would need to compute its global address, add the grid offsets, then convert back to find the owning node and offset to classify the position. These conversion operations are extremely expensive when repeated for each position, and we would like to use a more efficient method. With the data layout restrictions imposed in chapter 3—i.e. block decomposition—when an element at local offset i has a partner on node p at offset j it is fairly likely that the partner of the element at offset $i+1$ is on node p at offset $j+1$. If we can detect the runs where a local sequence corresponds to a

```
/* OMITTED: store sequence of noob inactive positions */
vp = noob;
/* Scale mulfact for effect of contiguous active seq in highest axis */
mulfact *= ub [nb-1] - lb [nb-1];
do {
  /* OMITTED: store sequence of mulfact active positions */
  vp += mulfact;
  k = nb-1;
  noob = oob [k];
  /* Iterate through loops, wrapping axes and adding OOB */
  while ((0 <= --k) && (++idx [k] == ub [k])) {
    noob += oob [k];
    idx [k] = lb [k];
  }
  /* OMITTED: store seq of min (noob, vplimit-vp) inactive positions */
  vp += noob;
} while (0 <= k);
```

Figure 6.4: RLE Storage Procedure for Boundary Contexts

src

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

s0

	0	1	2	3
0	?	?	?	?
1	0	1	2	3
2	10	11	12	13
3	20	21	22	23

s1

	0	1	2	3
0	1	2	3	?
1	11	12	13	?
2	21	22	23	?
3	31	32	33	?

Figure 6.5: Block-based Grid Sends

---

sequence on a (local or remote) node, we can treat them all identically, without needing to perform the address calculations at each position.

One way to do this is to specialize the grid communications functions by rank and type of shift. Figure 6.5 shows an example using a two-dimensional shape allocated on one node, moving one step along axis 0, and separately one step along axis 1. For the first case, the send can be accomplished with a single move of a twelve-element sequence from local offset 0 to offset 4; in the second, it is accomplished with four moves of three-element sequences. Though the example does not involve communication between nodes, distributed shapes permit similar behavior, sending whole blocks at once. By detecting at the start of the send which axis is being offset, sends along only one axis can be handled with routines optimized for these cases.

While such an approach works on the particular cases that are implemented, it does not improve the general case, and results in a large performance penalty when the user codes a grid operation which does not match one of the specialized versions: say, a shift along

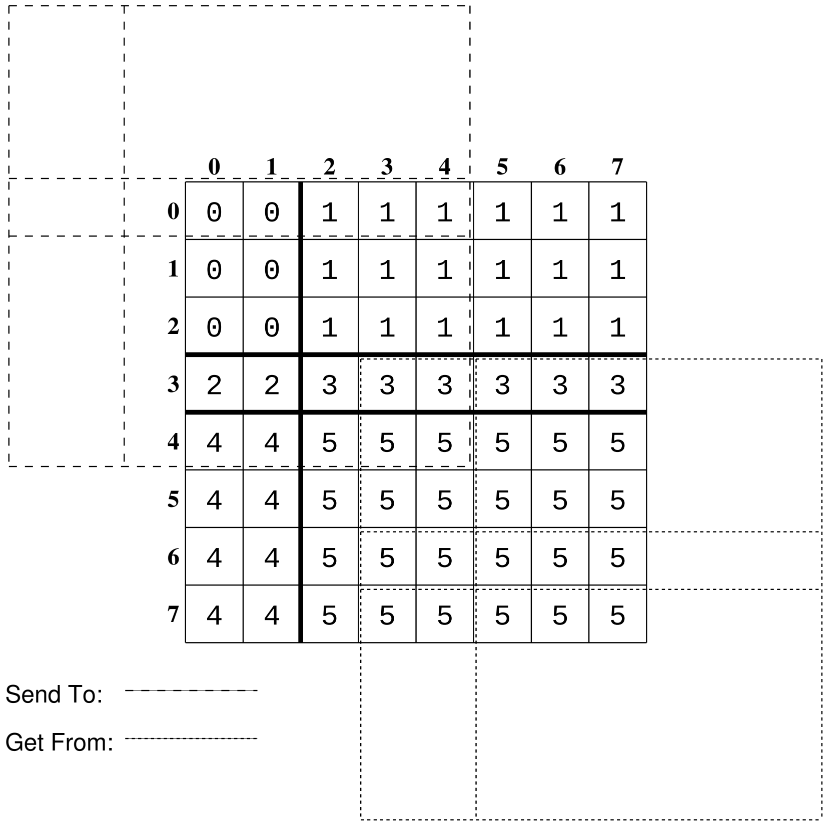


Figure 6.6: Example Grid Send Sequences: `[.-3][.-3]dest = src`

two axes at once. It is also necessary to make the system work when the source or destination variable is a field of a parallel structure (in which case the "chunks" are not contiguous and we must move them element-by-element), when send communications require a per-element operation, and with all the various actions desired on out-of-bounds access: reading fill elements, warning, aborting, or ignoring. The resulting code, filled with special cases, is a software maintenance nightmare, and poorly satisfies the C* programmer's needs. Therefore, we would much rather spend the effort designing a general case which is more easily verified, very near though perhaps not quite as efficient as the special cases, and performs equally well on the more unusual shifts along multiple dimensions simultaneously or on higher-ranked shapes. Such a general case implementation is the goal of this chapter.

A measure of the complexity of the problem can be achieved by examining the grid send depicted in figure 6.6. In this operation, none of the target positions exist where nodes 0, 1, 2, or 4 would send data, while node 3 sends data only to nodes 0 and 1, and node 5 sends data to all nodes. Nonetheless, all data can be interpreted as contiguous regions which are treated alike. These regions are listed in table 6.2, where the notation $n \rightarrow a@b$ indicates a sequence of n elements that is sent to offset a on node b.

To complete a grid send, each node must know what blocks it must send to which other

Node	Region Operations
0	6→oob
1	18→oob
2	2→oob
3	1→oob ; 2→0@0 ; 3→0@1
4	8→oob
5	1→oob; 2→2@0; 3→6@1;
	1→oob; 2→4@0; 3→12@1;
	1→oob; 2→0@2; 3→0@3;
	1→oob; 2→0@4; 3→0@5

Table 6.2: Region Decomposition of Grid Send in Figure 6.6

	2	3	4	5	6	7
4	b	0	0	1	1	1
5	b	0	0	1	1	1
6	b	2	2	3	3	3
7	b	4	4	5	5	5

Node 5 sends

	2	3	4	5	6	7
0	3	3	3	F	F	F
1	5	5	5	F	F	F
2	5	5	5	F	F	F

Node 3 receives

Figure 6.7: Nodal Region Info for Grid Send in Figure 6.6

nodes; an overlay indicating sent data is outlined with dashed lines in figure 6.6. Similarly, to detect that all data have been received, nodes must know what other nodes will be sending them data. This information is found by negating the offset vector, as is shown by the dotted outline. In both cases, the superposition of the inter-node boundaries of the shape onto the local subgrid breaks the local data into regions. Examples showing the send regions of node 5 and the receive regions of node 3 are in figure 6.7. The cells with numbers indicate the remote partner with which the cell communicates in the operation; those with value `b` in the send case would be sent out of bounds, hence are ignored; and those with value `F` in the receive case are ones where the corresponding sending positions are out of bounds, hence must be read from the fill variable.

In conjunction with the loop simulator discussed earlier, figure 6.7 gives the basic idea which allows us to recognize and treat sequences of similarly-handled data. Essentially, we emulate the loops which iterate over the positions in row major order, and detect whenever an axis index crosses over one of the darkened lines which indicates that the target position has stepped onto a different remote node. These *split points* are found for each axis by adding the grid shift offset for the axis to the axis indices and recording the places where the resulting offset crosses into another node, or out of the shape. The walk code which recognizes cross-overs is a straightforward extension of the loop emulator described in the previous section. Just as with boundary contexts, we do not have to step through the highest axis one element at a time: we can take the whole region between split points (at columns 2, 3, 5, and 8 for the node 5 send example) and deal with each region as a block. It is easy to determine the remote node, by adding and subtracting `distpprod` values (cf. section 3.2) as split points of an axis are crossed. By executing the loops and computing offsets for each region, this method yields exactly the sequences listed in table 6.2.

The sole complication is determining the remote offset to which the sequence should be sent. Doing this requires translating the global subgrid indices for the source sequence to local subgrid indices on the remote node, and computing the corresponding offset. Since both sequences are contiguous regions on their respective nodes, the difference between local and remote offsets is a constant for a particular sequence. By using index values that are global positions (rather than relative to the start of a particular node's portion of the axis), the translation is fairly straightforward. The offset of the currently indexed position on the local node is:[1]

$$o^{\text{local}} = \sum_{k=0}^{\texttt{rankof(S)-1}} \left((\texttt{idx}[k] - \texttt{DimAbove}^{\text{local}}[k]) * \texttt{NPA}^{\text{local}}[k] \right)$$

which performs the global-index-to-local-offset translation using information about the local subgrid. The offset on the corresponding remote node is:

$$o^{\text{remote}} = \sum_{k=0}^{\texttt{rankof(S)-1}} \left((\texttt{idx}[k] + \texttt{delta}[k] - \texttt{DimAbove}^{\text{remote}}[k]) * \texttt{NPA}^{\text{remote}}[k] \right)$$

Thus, given a local offset, we can compute the corresponding remote offset by adding $o^{\text{remote}} - o^{\text{local}}$ to it. For example, the sequence of three elements at offset 3 on node 5 that are to be sent to node 1 begin at global position $\langle 4,5 \rangle$. Translated by the grid shift $\langle -3,-3 \rangle$ this becomes global position $\langle 1,2 \rangle$, which is $\langle 1,0 \rangle$ in the local subgrid of node 1, corresponding to offset 6. Using the subgrid information from figure 3.4 in the above formula, we get:

$$o^{\text{remote}} - o^{\text{local}} = ((4-3-0)\cdot 6 + (5-3-2)\cdot 1) - ((4-4)\cdot 6 + (5-2)\cdot 1) = 6-3 = 3$$

The difference value varies depending on the sequence location and remote node, and can be simplified to be a linear function of the index values. The scaling factor and constant term of the resulting linear functions would be different for each axis and remote node. We could dynamically allocate a table to hold the scaling factor and constant offset, but feel that the calculation time savings in so doing (a couple integer arithmetic operations) is not worth the additional code complexity.

While this walk to find sequences of each type (local, remote, out-of-bounds) could be done in one pass with some complication to control flow, grid communications tend to partition the shape into fairly long blocks of data, each of which is associated with only one

1. We use $\texttt{NPA}^{\text{local}}[k]$ as shorthand to represent the `num_per_axis` value for the kth axis of the current shape with respect to the local node's distribution; similarly for `DimAbove`, and the remote node's distribution information. Refer to section 3.2.1 for details on distribution parameters.

operation (send off-node, move locally, read from off-node, copy from fill, ignore). The cache advantages of a linear access pattern are less significant in this case, but performance will be adversely affected if data which are sent off-node are found and sent only in the last stages of the operation. Therefore, grid communications routines are the exception to our rule favoring linear walks through data, and we perform several consecutive walks, first gathering and sending off-node data (to take advantage of latency), then performing local moves and fills (or bounds checks), and finally storing data from off-node into their desired local area. The last step can be handled implicitly by using the communications handler infrastructure described in chapter 4: on entry to the communications routines, we register a function which reads the incoming data and places them in the appropriate location.

The need to perform multiple passes through the data when searching for regions of a particular type can have undesirable performance effects, since in each pass we want to find only a subset, often small, of the regions. Using the example of figure 6.7, if we are looking only for regions which are stored on the local node, we will examine and reject eleven regions in node 5 before finding the one we want. In the general case, each pass would require examining every region and picking those that are associated with our node, or any remote node, or an out-of-bounds area. However, by modifying the portion of the loop emulation control flow that determines when index values need to wrap, we can drastically reduce this overhead.

Consider in particular the send regions for node 5 when scanning for local data (the single region going to node 5). We necessarily start at the beginning of the shape, at global position $\langle 4,2 \rangle$. The key point to note is that, if we complete the scan of row 4 without finding a region of interest, we need not examine row 5 because the types of regions encountered will be the same as the previous row, until we cross over the next split point along axis 0. Therefore, once we've rejected all regions at a given axis, we can skip immediately to the next split point on a lower axis rather than iterate through it index-by-index. When scanning for a particular region type which does not appear between given split points, this reduces the number of regions examined from being on the order of the dimension of an axis to the order of number of distribution partitions along the axis.

The details of the implementation for grid communication, considering all these issues, are too complex to present in the body of this dissertation. The source code for grid-based communication, which includes both get and send, and a modification of the split-point handling which allows torus get and send (out-of-bounds references wrap around the shape), consists of 3500 lines of heavily-commented C code. A subset of this code corresponding to a grid send operation, including code to identify sequences and walk the different classes of sequence, is presented in appendix A. The comments in the code, in conjunction with the high-level overview presented here, should provide the determined reader with enough information to implement these operations in her own runtime system.

6.3 Evaluation and Related Work

We are unaware of any other general approaches to optimizing runtime resolution of regular communications. For many years, systems for parallel languages, especially derivatives of Fortran, have been able to detect certain cases at compile-time and generate calls to send and receive data, but these are most often restricted to times where the cluster size, shape size, and shift values are known at compile time.

Kali (Koelbel, 1990) detects, at compile time, communications patterns that are a superset of our grid communications operations, and emits code to compute sets of positions that are to be sent to or received from other nodes. The general framework described in (Koelbel, 1990) supports both block and cyclic distributions, and could be extended to multiple dimensions. The representation of elements as sets rather than as sequences implies that the data structures and emitted code for multi-dimensional distributions and shifts such as that in figure 6.6 would be highly complex. Though the Kali framework is relatively independent of shape dimensions and the size of the target cluster, it does require knowledge of shape rank at compile time. It is not clear that the framework could be reasonably extended to a completely general runtime system that was rank-independent; on the other hand, it is not clear that Kali's target applications would require such an extension.

Experimentation using a specially-instrumented grid send implementation within pC* indicates that the cost of runtime resolution is fairly low. Performing grid sends on a two-dimensional shape with 2048×2048 positions on clusters from 1 to 12 nodes, the initial step which builds the loop bounds is on average 0.21% of the total communication time, with a maximum of 3%. For communications which had more than one in-bound local sequence (e.g., one per row), an average of 4% of the communication time went to overhead in computing sequence locations and remote node information; when there was only one such sequence, the overhead was less than 0.1%. These overheads are sufficiently low that we feel the runtime mechanism described here is likely to be quite competitive with a compile-time implementation such as Kali's. When dimensions or shift offsets must be specified at runtime, both systems perform much the same calculation. Our system could easily be adapted to save the sequences of each type as a communication schedule, and could be integrated with Kali or the schedule-based mechanisms for general communication described in section 5.5.

Another measure of performance is to compare grid communications with a general communication which involves the same communications pattern. For this experiment, we used the torus version of the grid routines to implement a shift along the (1,1) diagonal of a rank-2 shape. Figure 6.8 shows the communications pattern for a shift of half the shape size in each axis.

We compared the time to perform the shift from a source pvar into a destination pvar on clusters from one to twelve nodes and a shape with 1024×1024 four-byte elements.[2] Three

2. Had we used one-byte elements as we did in the previous chapter, the effect of location overhead would taint the comparison: general send has a 4-to-1 overhead when using one-byte elements, while four-byte elements have 1-to-1 overhead.

orig

	0	1	2	3
0	0	1	2	3
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

shifted

	0	1	2	3
0	22	23	20	21
1	32	33	30	31
2	2	3	0	1
3	12	13	10	11

Figure 6.8: Torus shift used for grid versus general communications comparison

different times were measured:

- The time to copy the data from the source pvar to the destination pvar with no communications routine involved. This yields a lower bound for any communication operation.
- The time to perform the shift using the torus adaptation of the grid mechanism described in this chapter.
- The time to perform the shift using left-indexed general send, as described in chapter 5.

The results are presented in table 6.3. Times are in seconds, for one send of each type, and are the median of five experiments, except for copy, which is the median of the fifteen experiments for the given cluster size.

Three shift amounts are given for each cluster size. A shift of 0 is intended to measure the overhead involved in merely invoking the communications, even though it never requires exchanging data with another node. In this case, the grid routines perform the initial loop construction, then notice that no shifts will occur and simply copy the data. The Torus/Copy ratio column indicates that the overhead of merely invoking grid communications and building the loop bounds is less than the timing measurement noise (hence the incongruous result that it is apparently often faster to perform a zero-offset torus shift than copy a variable). However, the general routines still incur a cost, both in converting addresses to perform the moves and in global synchronization to determine that no data will be arriving from off-node.

A shift of one position along each axis is a more common case. Since we used the default row-distribution for shapes, each node writes values to the node "below" it, and reads them from the node "above". The torus operation on 2 and 4-node clusters indicates that the operation is only 5% more expensive than copying the data with no communication. This combines the overhead of determining the moves with the time required to transmit data between nodes. It is likely that the portion of overhead attributable to the grid mechanism is smaller than 5%: with larger clusters the time required to perform the operation remains constant at 60msec, indicating communications overhead or latency is the primary consumer of time. The difference between grid and general communication is apparent, with the general mechanism taking about ten times longer, except where the torus had reached its network-limited lower bound.

Cluster Size	Shift Amount	Copy Time	Torus Time	General Time	Torus/Copy Ratio	General/Torus Ratio
1	0	0.171	0.170	2.036	0.995	11.948
	1	0.171	0.250	2.049	1.463	8.183
	512	0.171	0.179	2.034	1.047	11.342
2	0	0.087	0.086	1.285	0.989	14.885
	1	0.087	0.092	1.294	1.057	14.025
	512	0.087	3.697	7.320	42.354	1.980
4	0	0.056	0.056	0.706	1.000	12.614
	1	0.056	0.059	0.717	1.050	12.201
	512	0.056	1.881	3.682	33.593	1.957
8	0	0.015	0.014	0.419	0.919	30.831
	1	0.015	0.060	0.428	4.054	7.130
	512	0.015	0.951	1.867	64.270	1.963
12	0	0.008	0.007	0.324	0.843	46.329
	1	0.008	0.060	0.337	7.205	5.629
	512	0.008	0.672	1.295	81.012	1.926

Table 6.3: Grid versus General Communication comparison. Times in seconds to send 2^{20} four-byte values.

The third case performs a shift half way along each axis. This requires all data to be sent off-node in both mechanisms. Here we can see that the torus and general communications are within a factor of two of each other. This is exactly what one would expect in the case where the network performance is the sole determining factor. Recall that in general communications each element sent off-node has an associated offset sent along with it. This means that, for a four-byte element, eight bytes are sent for each position. With the grid algorithm, we send packed sequences consisting of an offset, a count, and a run of count elements. Sequences for the 512-position shift are 512 elements long, so the amount of data sent in the torus operation is slightly more than half the amount sent in the general communication. This means that the best we should expect of general communications is that it be $2\times$ slower than torus communications for these patterns. As shown in section 5.5, with high communications behavior, even the overhead of general communications is overwhelmed by network latency.

In section 6.2 we mentioned that certain grid communication operations can be recognized and handled as special cases. The initial version of pC*, based on the University of New Hampshire C* compiler (Lapadula & Herold, 1994), did this, with special cases for one and two-dimensional grid get operations, and a general algorithm which performed pointwise address calculation for other cases. As time went on more cases became important: in particular, send operations, which do not have the inherent two-phase structure of get operations, are heavily used in code originally written for the Connection Machine.[3] Torus operations are also used in several algorithms, and to prevent a large and confusing performance discrepancy between grid and torus operations, these should also be special-cased. As time went on, the grid/torus implementation module became a morass of conditionals and questionably reliable code. The final straw was the support for general block distributions described in section 3.2, which invalidated several assumptions in the special-case code. To clean up this maintenance and reliability problem, we designed the algorithm described in this chapter. However, it is still natural to question whether a special-case implementation is sufficiently faster to be worth maintaining for the most common cases.

To address this, we compared pC* with a more recent implementation of UNH-C*,[4] on both sends and gets with one-position shifts in various directions using one, two, and three-dimensional shapes. To avoid bias due to differences in the network implementation of each system, we ran the tests on a single processor. The results, with times in seconds on a SS20/612, are presented in table 6.4, and in graphical form in figure fig:gridops:eval:pcsvsunh.

We started by determining the base time to copy data from the source value to the destination. Both systems implement this with a VP loop performing the assignment to each active element in turn. In contrast, calling a routine which performs the same copy but over the entire shape at once, rather than element-by-element, runs twice as fast. This routine is only implemented in pC*, but is essentially the operation performed by the torus shifts of 0 positions in table 6.3, and on sequences of data in both pC* and the UNH optimized routines.

Following this in table 6.4 are the performance results for get and send operations for all three shapes. The shapes all had 7529536 one-byte elements, and were allocated to be "square": i.e., shapes were 7529536, 2744×2744, and $196 \times 196 \times 196$. One key point in understanding the results is that get operations store their results in a compiler-allocated temporary, which is then copied into the active positions of the `dest` variable by elementwise assignment. This is because in most cases the results of the get are used from the compiler temporary in a complex expression; if only the communicated results are desired, a C* send communication would generally be coded instead. Neither compiler recognizes the opportunity to store the results directly in `dest`, though this could be done in both systems, modulo some complications related to context.

Taking this (roughly 0.5sec) overhead for get operations into account, we can see that, in pC*, grid get and send have the same performance when all communication is intra-node. Communications which do not involve shifts along the highest (right-most) axis are generally slightly faster than those that do, because the sequences of values operated on are longer

3. Although in the absence of context a grid get may be implemented in a single phase by interpreting it as a send in the reverse direction, when context is involved this can result in performance problems. Since context is determined on the requesting node, pre-emptively sending data may result in higher communication costs than a request/reply implementation. For small enough shapes, the advantage in avoiding one communications phase may be worthwhile.

4. Version 950609.

UNH	pC*	UNH/pC*	Operation
0.517716	0.55136	0.93898	`dest = src`
n/a	0.3062755	n/a	`memcpy (&dest,&src,boolsizeof (src))`
1.535910	0.8570395	1.79211	`dest = [.-1]src`
53.620190	0.365047	146.886	`[.-1]dest = src`
0.948832	0.850059	1.1162	`dest = [.][.-1]src`
0.868239	0.828215	1.04833	`dest = [.-1][.]src`
0.870376	0.8391835	1.03717	`dest = [.-1][.-1]src`
73.510028	0.331552	221.715	`[.][.-1]dest = src`
73.589463	0.31381	234.503	`[.-1][.]dest = src`
73.519656	0.336797	218.291	`[.-1][.-1]dest = src`
92.688201	0.995749	93.0839	`dest = [.][.][.-1]src`
92.809967	0.91927	100.961	`dest = [.][.-1][.]src`
92.684462	0.846993	109.428	`dest = [.-1][.][.]src`
92.777239	1.010195	91.8409	`dest = [.-1][.-1][.-1]src`
94.027049	0.5133385	183.168	`[.][.][.-1]dest = src`
94.060914	0.297252	316.435	`[.][.-1][.]dest = src`
94.021485	0.3156275	297.887	`[.-1][.][.]dest = src`
94.076640	0.4722375	199.215	`[.-1][.-1][.-1]dest = src`

Table 6.4: pC* General Grid versus Special Case code. Time in seconds to send 7529536 (196^3) one-byte values on a single-processor.

in those cases. This also holds for the specialized implementations in UNH C*. There are a variety of interesting conclusions that can be drawn from the results:

- The grid algorithm described in this chapter is not particularly sensitive to the rank of the data being operated on, or the number of axes along which shifts occur simultaneously. The performance differences observed can be explained by differences in the average length of a sequence.
- The general algorithm in pC* is faster than the special-case code in UNH-C*. In the case of one-dimensional operations, the difference is nearly two-fold. It is not immediately clear why this should be so, since in the case of single-node clusters both systems perform a small amount of pre-computation prior to a call to `memcpy`.
- The general algorithm in pC* is well over 100$\times$ faster than a general algorithm which performs address computation at each position.

The memory requirements of both systems to execute these tests are also informative. pC* has a peak usage of roughly 31MB, with about 24MB resident at peak. The usage breaks down into 7.5MB for each of `src` and `dest`, 7.5MB for the compiler temporary used in the get assignment, and 7.5MB for the context build arena (which was not used in the test

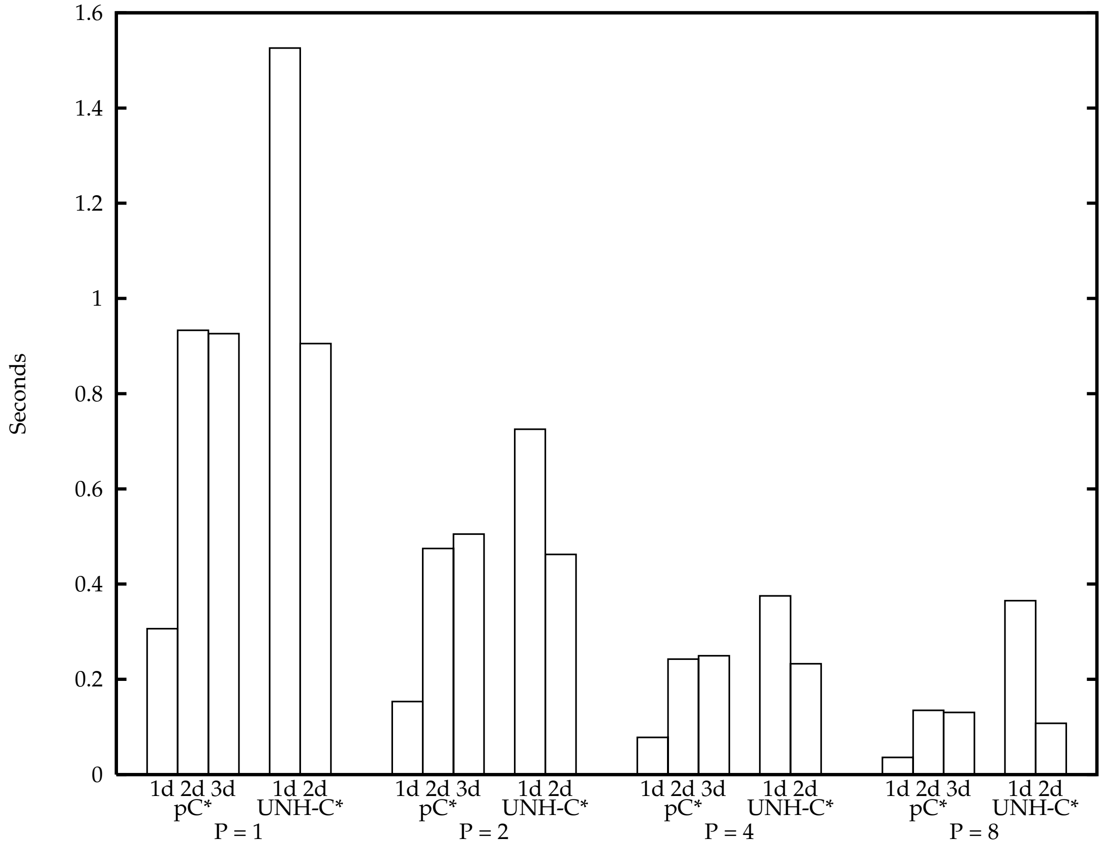


Figure 6.9: pC* General Grid versus Special Case code. Time in seconds to send 7529536 (196^3) one-byte values, 1, 2, 4, and 8 processors.

---

program). In contrast, the UNH-C* program had a peak usage of 119MB, most of which was listed as resident memory throughout execution. It is unclear where the memory was being used or what effect this had on the performance of UNH-C*, though no swapping occurred during the execution (the benchmark machine had 256MB of physical memory).

CHAPTER 7

EVALUATION OF PC*

The proof of the pudding is in the eating.

— Miguel de Cervantes, *Don Quixote de la Mancha, bk. IV, ch. 10, p. 322*

In which we walk the walk. We evaluate pC on a set of eight benchmarks: four designed to test the operations described previously in this dissertation, two image processing problems, and two general problems. We prove portability by giving results on a cluster of Ethernet-networked SPARCstations, a multiprocessor SGI, and an Intel Paragon. Efficiencies (speedup per processor added) range from 45% to 90% on the primary target platform. When contrasted with optimized sequential C implementations on the same hardware, pC* outperforms C on two benchmarks with as few as two processors, and on three more with as few as four processors. When compared with the Thinking Machines Corporation implementation of C* on a 64 node Connection Machine CM5, pC* on a twelve SPARCstation Ethernet cluster outperforms the CM5 in real terms on three benchmarks, due in large part to the optimizations described in this dissertation. On a per-processor basis, pC* outperforms the CM5 on six of the eight benchmarks.*

7.1 Target Platforms

Evaluation tests were performed on a total of six architectures using pC*, as well as two architectures using other C* implementations. The number of processors available for pC* testing ranged from one to twenty-four. The particular platforms that will be referenced in this chapter are:

the cluster The primary computational cluster for pC* is a network of twelve Sun SPARCstation 612s, with two 60MHz SuperSPARC processors each, 1MB of external cache per processor, and 256MB of memory per machine. The machines are connected with 10baseT Ethernet (10Mbps) in a star network through a Kalpana 2015 RS EtherSwitch. The host operating system was Solaris 2.3, and communication was performed through TCP sockets (`AF_INET`, `SOCK_STREAM`). Compilations were performed with GNU gcc version 2.6.3, using flags `-O2 -DNDEBUG -msupersparc`. The cluster is operated by Oasis Research Center, Inc.

the SGI To determine performance on a different interprocessor model, we used a Silicon Graphics 4D340, with four 33MHz MIPS R3000 processors, primary data and instruction caches of 64KB each, a secondary data cache of 256KB, and 64MB of physical memory. The worker processes communicated through the System V Message Passing (`msgctl`(2)) interface under the host operating system, Irix 5.2. Compilation was performed with the SGI ANSI C EOE version 3.18 compiler, using flags `-O2 -DNDEBUG`. The SGI is operated by the Department of Computer Science at the University of Arizona.

the Paragon As a second check of portability and other IPC interfaces, we used an Intel Paragon XP/S Model A4, with 16 compute nodes each with two 50MHz Intel i860 processors (one reserved for network management), and 16MB of memory per node. The host operating system, Paragon OSF/1 Release 1.0.4 Server 1.3, consumed approximately 8MB of memory on each node, drastically limiting the problem sizes which could be run. Worker processes communicated using the built-in Intel NX message passing library over the Paragon's 30MBps mesh network, with 1000-byte messages. Compilations were performed with GNU gcc version 2.6.3, using flags `-O2 -DNDEBUG`. The Paragon is operated by the Department of Computer Science at the University of Arizona.

the CM5 We compared the system with the defining implementation of C* by Thinking Machines Corporation, on a set of Connection Machine CM5s with 64, 256, and 512 nodes (Hillis & Tucker, 1993). Each node consists of one 32MHz Sparc chip coupled tightly with a specially-designed four-processor vector unit, and has 32MB of memory available on each node. The nodes are connected through a 20MBps fat-tree network. Compilations were performed with TMC C* version 7.2, using flags `-O2 -DNDEBUG`. The CM5 is operated by the Army High Performance Computing Research Center at the Minnesota Supercomputer Center in Minneapolis, Minnesota.

7.2 Target Applications

Though there are a variety of applications that can be used to measure the performance of the pC* system, we have chosen a total of eight. To provide an honest evaluation of the performance of pC* (Sahni & Thanvantri, 1996; Bailey, 1991), we tested both C* and C implementations of these, in each case with an implementation that a reasonably skilled programmer would consider to be appropriate to solve the problem. Here we describe the benchmarks and their C* implementations. The C* source code for the benchmark programs is presented in appendix A, and contains more details on the algorithms used.

The first four problems were chosen specifically to exercise components of the library that were described in the previous chapters.

fft A straight-forward butterfly implementation of the Fast Fourier Transform using complex floating point numbers. The normal divide-and-conquer FFT algorithm is parallelized using a loop with $\log(n)$ iterations performing pairwise combinations of 2-element FFTs, using general send to exchange the operands between paired virtual processors. The test performs the transform, and then the inverse transform. This benchmark exercises general communication with no collisions.

histeq Histogram equalization of a digital image with eight bits per pixel. The intensities in the image are histogrammed using the method introduced in section 2.1.2, then the bin values are used to spread the image range over the pixel range, and the equalized pixel values are read back into the image shape. This benchmark exercises general communication with high collision behavior.

njac A standard Jacobi iteration program. Boundary and internal elements are initialized, then 100 iterations of four-point-stencil (North-East-West-South) averaging is performed over the internal region. This benchmark exercises grid read.

roadnet The program is given a map with certain pixels distinguished as belonging to a "road". Pixels adjacent to roads are set to their distance from the road, using contextualized grid send of the adjacent four elements with a minimization operation. The test is initiated with "roads" which superimpose an "X" reaching to each corner and a "+" to each edge, centered on the middle of the map. By setting the active context to the most recent road perimeter only, this benchmark tests operations with a highly inactive context.

The C code implementations of three of these four programs are highly optimized, as will be discussed in section 7.6, and comparisons between the C* and C versions do not give a reasonable representation of the system's performance on more complex programs. We evaluated four additional benchmarks, which do not necessarily exercise components of pC* that were described in this dissertation, but also represent the type of operations that are common in our real target applications and provide a better representation of the pC*'s capabilities. These include:

amp An image amplitude screener. The value of each pixel is compared with the average value of its surrounding 8 pixels, and a boolean pvar is set to indicate the pixels which are above some threshold percentage of their surrounding pixels. This benchmark primarily exercises the prefix scan operation, though it also performs grid sends.

julia Essentially a Mandelbrot set calculation. This is an "obscenely parallel" benchmark; no communication takes place in the computation loop. The Mandelbrot set develops large inactive regions, and block distribution leads to load imbalance. Therefore we precede the computation loop with a general send which distributes the computation field in a cyclic manner, as described at the end of section 5.5.

mm The straightforward C* implementation of matrix multiply. Working on square matrices only, we start by transposing one of the operands using general send, then use the extended library routines `copy_spread` and `reduce` to spread each column in turn across a parallel value, do a point-wise multiply, then reduce each row into the appropriate column of the result.

rf An image rank filter. The value of each pixel is replaced with the median of the values in the 3×3 window surrounding it. This benchmark exercises torus grid sends, and inlined context.

Each of these tests was run on data sizes ranging from a few thousand to roughly thirty-two million elements, one per virtual processor, with larger data sizes skipped on architectures which lacked sufficient computational capacity or memory to run them. The experimental method was to run each benchmark over its range of data sizes on all subsets of a particular architecture, then repeat this four more times. The measured time for any multi-node run was the maximum elapsed time observed on any node; some nodes may have finished more quickly, if work was not evenly distributed, but since the problem is not solved until all nodes are finished, the maximum elapsed time measures the real solution time. Performance results in this chapter are the median of the resulting five runs for each benchmark–host cluster–data size.

7.3 Performance of pC* On the Cluster

We ran the benchmarks on the cluster, with 1, 2, 4, 8, and 12 workers, running only one worker on each machine.[1] The raw performance is shown in the graphs in figure 7.1. Throughout this chapter, the legends in the graphs indicate the host architecture (e.g, "cl1" for the SS20 cluster, "Fd340" for the SGI 4D340, "pgon" for the Intel Paragon), with the number of processors involved appearing in parentheses following the host code (e.g., "cl1(4)" for a four-processor SS20 cluster).

Tests histeq, amp, and julia appear to have consistent performance for a given cluster, once they reach their optimal operating size. fft implements an $O(n \log n)$ algorithm, so the slight decrease in capacity as size increases is appropriate; similarly for the $O(n^{3/2})$ algorithm used by matrix multiply.[2] The roadnet benchmark continues to increase, again as we should expect, since the number of road elements in the test increases with the lengths of the sides of the map, not the total number of pixels in the map.

rf performs well on small to medium problems, then drops to a lower plateau on larger problems. This behavior, which is more obvious in the SGI results to follow, is explained

1. Experience showed that using both processors resulted in inconsistent performance across different benchmarks, due perhaps to contention for the network interface and the memory bus, and the difference in in-kernel control flow when transmitting packets between two workers on the same machine, and between two workers on different machines. Testing with one processor disabled indicated that having the additional processor present and unused did not affect runtimes significantly, except for the smallest data sizes where the newly forked worker could execute simultaneously with the daemon which controlled distributed execution.

2. Note that the data size $n = t^2$, for 2d matrices with side t—for n an odd power of 2, we chose a t such that $|n - t^2|$ was minimized. When times are divided by square-root powers of the scaled size to correct for the algorithm time complexity, the performance curves become flat, as expected.

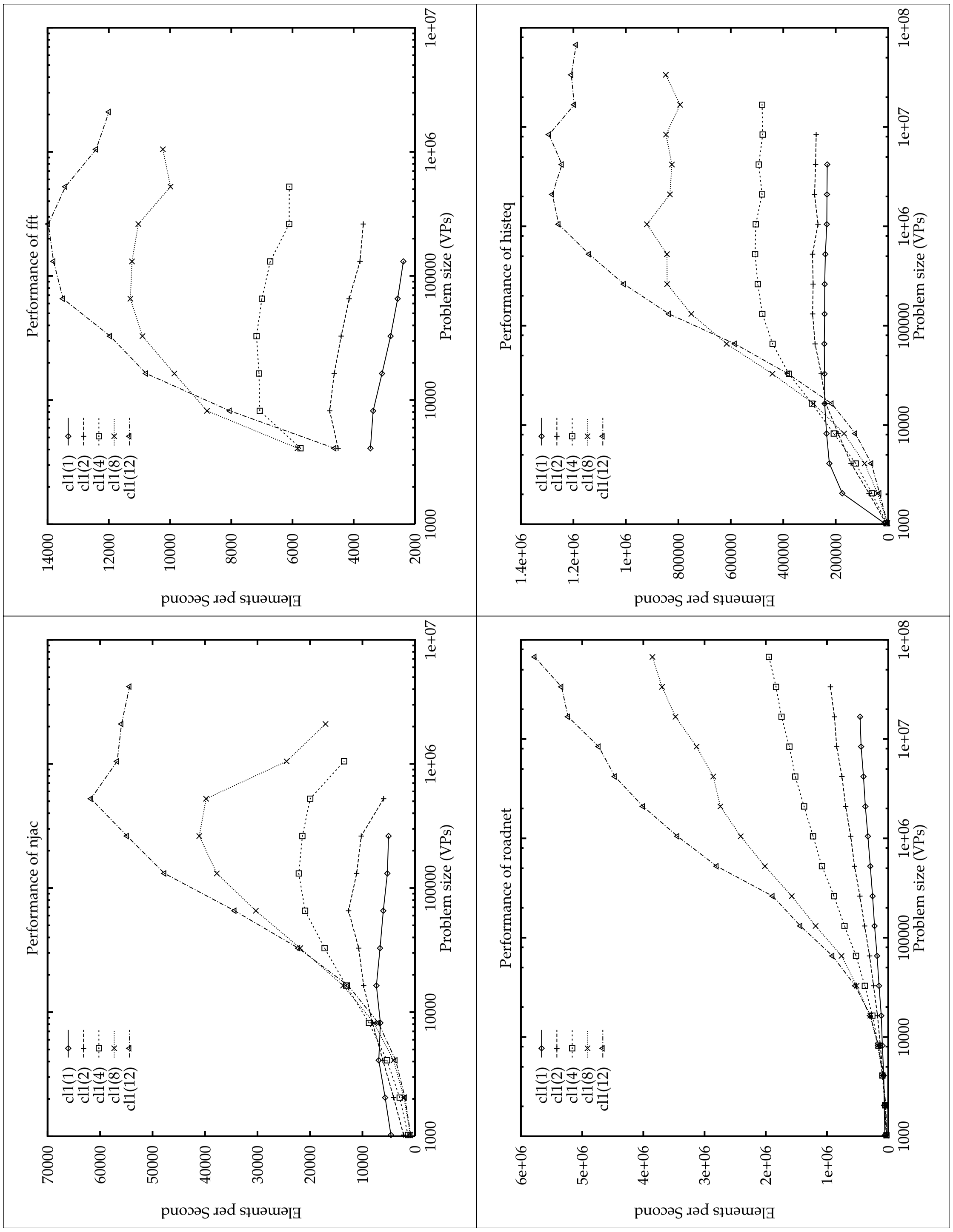


Figure 7.1a: Cluster Elements-Per-Second (Part 1)

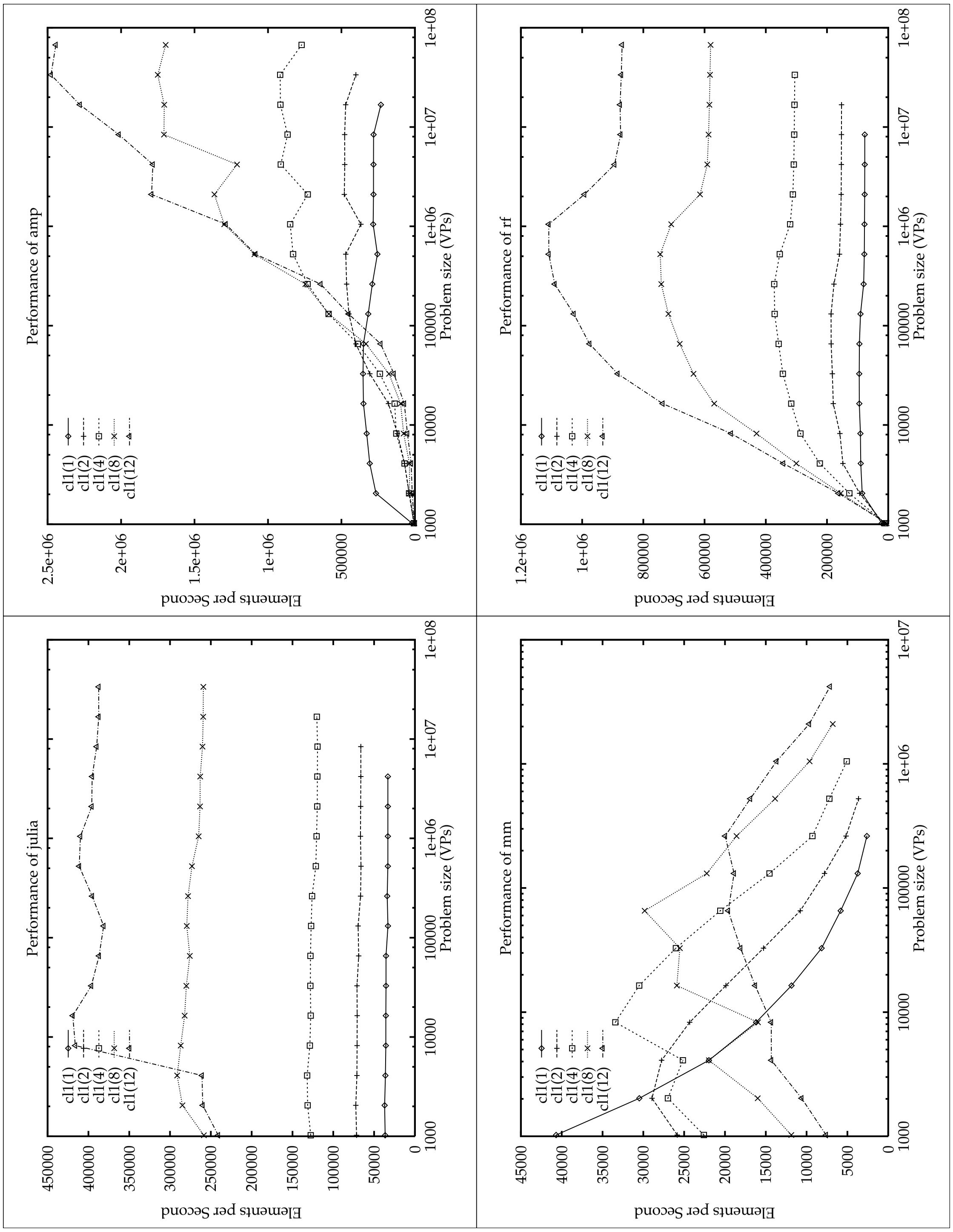


Figure 7.1b: Cluster Elements-Per-Second (Part 2)

by noting that, as problem size increases, the working data will no longer fit in the cache. The curves in rf indicate that if peak performance for size n is reached with a cluster with P nodes, peak performance for size $2n$ is reached with a cluster with $2P$ nodes, maintaining a consistent amount of memory per-processor.

One might expect njac to have a flat performance profile since the algorithm is linear in the number of elements, but in fact it takes a fairly steep dive at larger problems. Part of this is due to the cache thrashing also seen in rf, but the remainder, especially the large drop in performance with eight nodes at 2^{20} elements, is more significant. Runs of this particular benchmark consistently had significantly higher idle times than the same program and data size on other configurations—in fact, the same problem run on seven nodes executed in half the time required on eight nodes. The njac implementation performs a global reduction on each iteration, to check convergence and the need for an additional iteration.[3] Initial evidence indicates that the slowdown is due to differences in perceived latency using the LOGLOG reduction algorithm from chapter 4: the master node spends one tenth the time in the reduction than does node $P/2$. Further analysis is required to determine why this is occurring and an appropriate fix, and whether it occurs to the same degree when a better reduction algorithm is used (cf. page 99). We have found the Solaris 2.3/Ethernet platform to be susceptible to bad behavior with certain communications patterns,[4] and we currently believe the bad performance on reductions is due in part to latency effects in the fan-in and fan-out stages. A plausible solution is to use a different master node for each reduction, distributing communications patterns more evenly.

Other apparently anomalous results are the small dip at 512K with fft on eight nodes (experimental noise; not visible in a second set of experiments); the peak in histeq at 1M on eight nodes (experimental noise); the dip in amp at 4M on eight nodes (idle times in calls to `reduce`); and the performance drops for mm at 64^2 elements with four nodes and 91^2 elements with eight nodes. This last seems due to bad handling in Solaris Ethernet code of the communication pattern that arises from these problem sizes; idle times are increased, due perhaps to the fact that mm is the only benchmark that sends significant amounts of data in an operation that does not have a registered handler (cf. chapter 4). The performance drops for these problems are regularly reproducible using the Kalpana EtherSwitch, but are not visible when the same systems are linked with a Fore Systems ATM switch.

More interesting is the measure of speedup we get by increasing the cluster size. This comparison is shown in figure 7.2. As expected, many of the programs perform worse with

3. The benchmark actually ignores this value, since the benchmark is expected to iterate a specific number of times and not drop out early, but a real-world program would require it.

4. For example, the torus version of the grid routines in chapter 6 sends data to a node in one direction, and reads data from a different node. It is twice as fast, in terms of elapsed time, to send the data one element too far, then send it back one element, than to send the data exactly the distance required. The difference is consumed in system idle time during the single-phase communication. Using a different interprocess communications package (e.g., the System V Message Passing Facility) on the same hardware, the two-step send is two times slower than the one-step send, as we would normally expect. The only explanation we have for this is that the paired calls result in a communication pattern where each node both sends to and receives from both neighbors, and the Solaris TCP execution path is optimized for such exchanges and suffers when unequal exchanges occur.

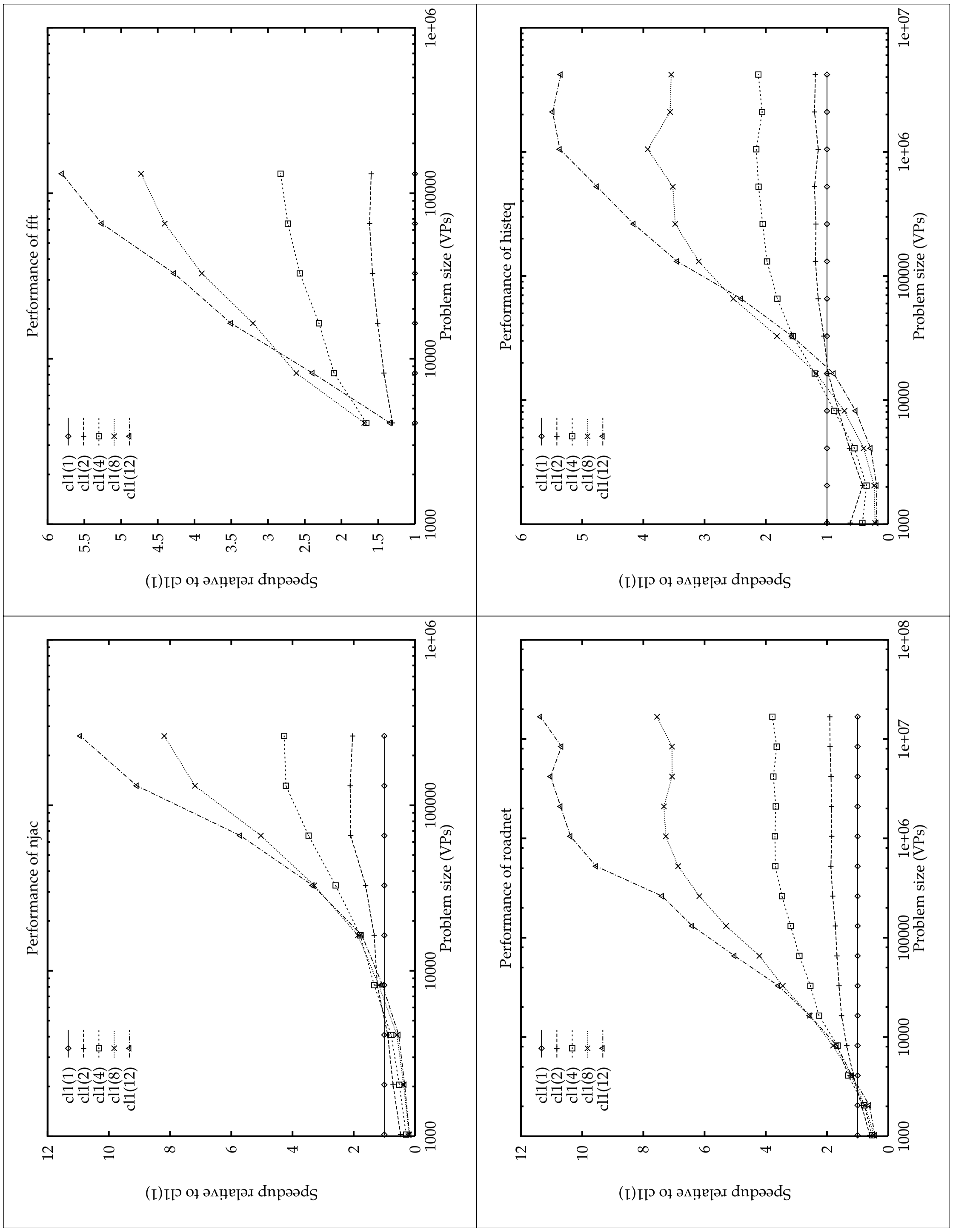


Figure 7.2a: Cluster Speedup (Relative to pC*−1 processor) (Part 1)

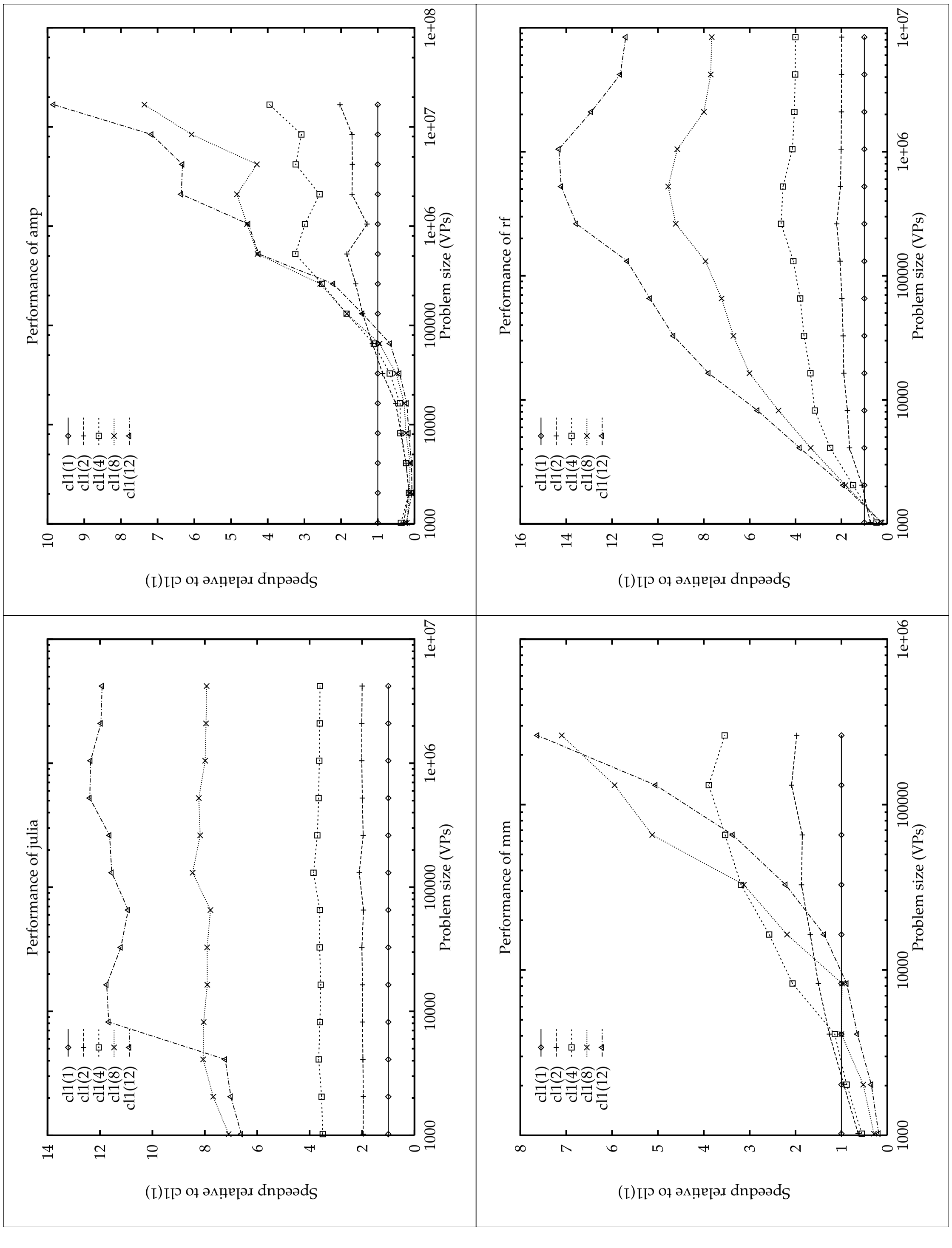


Figure 7.2b: Cluster Speedup (Relative to pC*–1 processor) (Part 2)

larger clusters for the smallest data sizes: the overhead of communication synchronization overwhelms the speedup from multiple processors. This is most clear in the graphs for amp and histeq. julia does not have communication in the timed loop, so does not suffer this overhead, and the smallest size used in fft is large enough that the overhead is absorbed.

A more clear understanding of scalability can be obtained from the efficiency graphs in figure 7.3. The efficiency is simply the speedup divided by the number of processors in the cluster. Most of the tests seem to converge toward a constant efficiency dependent on the program; roughly 45% for histeq, and closer to 95% for roadnet. The efficiency of fft is affected more strongly by cluster size than the other programs, which do not stress network communications so much. Super-linear speedup (efficiencies in excess of 100%) is observed for three programs at several problem sizes, occurring when the larger amount of cache memory allows faster execution for a particular problem size than did smaller clusters.

7.4 Performance of pC* on the SGI

The benchmarks were run on the SGI machine with 1, 2, and 4 processors. Raw performance in elements-per-second is shown in figure 7.4. The major difference between these results and those of the cluster is the clear effect of small cache memory on problems which require large amounts of data. Trends for each benchmark follow those of the cluster, until a certain (benchmark-dependent) data size is reached, at which point performance drops precipitously to a new plateau. The failure point doubles as number of workers (hence, total cache size) doubles. The efficiency results in figure 7.5 are affected by this, where super-linear speedup is observed in all tests except histeq; the efficiency generally exceeds 100% around the data size where the single-worker test exceeded its cache size and slowed down. Histogram equalization, which has a relatively small memory load per virtual processor, and does not re-visit data frequently, does not suffer as much from the small cache, and efficiency measurements are roughly the same for the SGI as they were on the cluster. The roughly 100% efficiency of fft, in contrast to its consistently decreasing behavior on the cluster, is explicable by noting that fft exceeds the first level cache with its smallest data size, and rapidly exceeds the second level cache as well. It is plausible that in the baseline one-processor runs, the bulk of time is spent waiting for data to be loaded to a cache level where the code may read them, and adding processors allows the memory loads to be pipelined, distributing the latency equally amongst the processors.

7.5 Performance of pC* on the Paragon

The performance results on the Paragon are shown in figure 7.6 and exhibit perhaps the cleanest curves of any test architecture. Most interesting is the performance of histeq at the largest data sizes that we even attempted to run, where not just cache but also physical memory was exceeded during execution, resulting in a drop to a constant performance limited by the access speed of the paging device. Clearly, execution of real-world problems requires systems with more than toy amounts of memory.

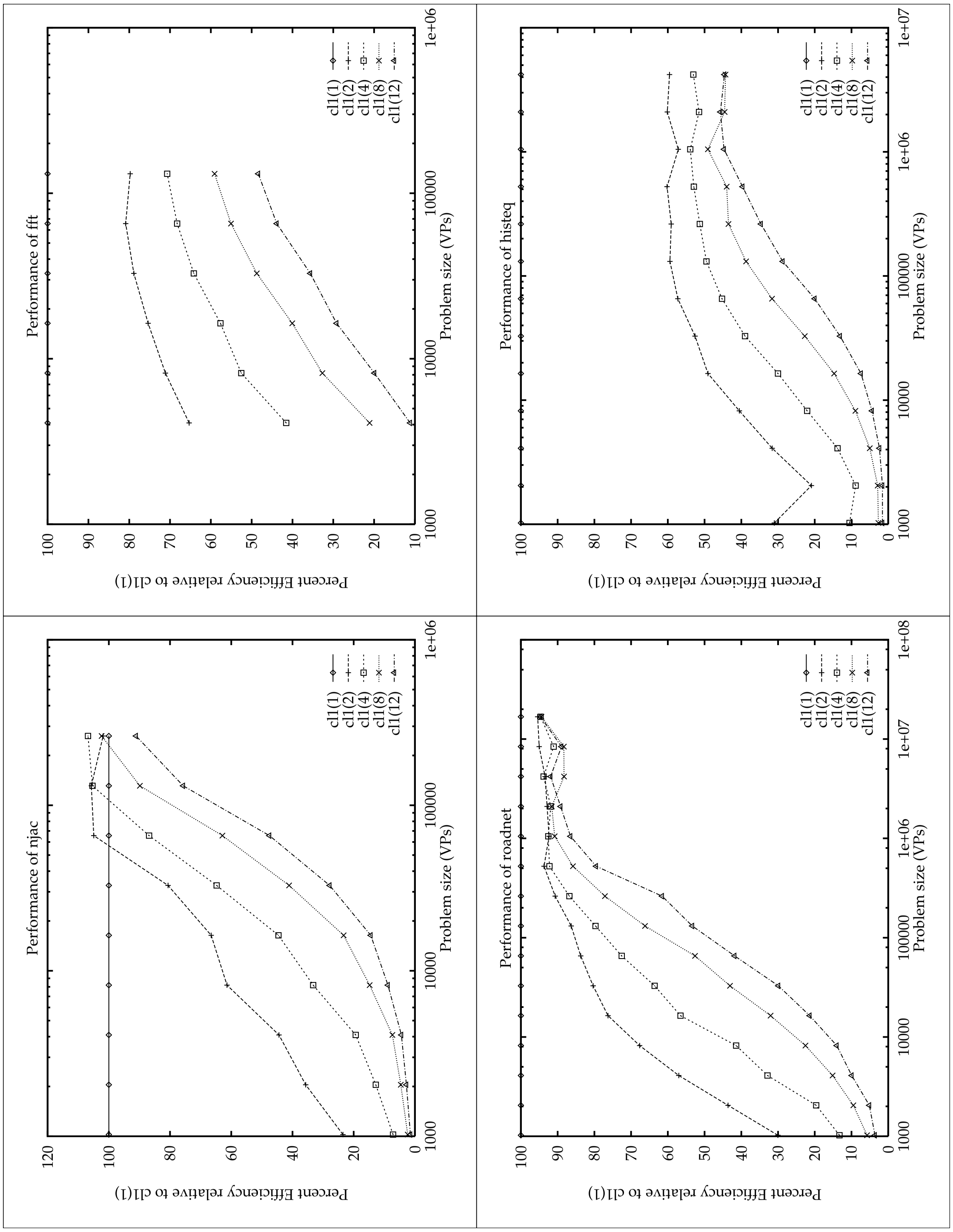


Figure 7.3a: Cluster Efficiency (Relative to pC*–1 processor) (Part 1)

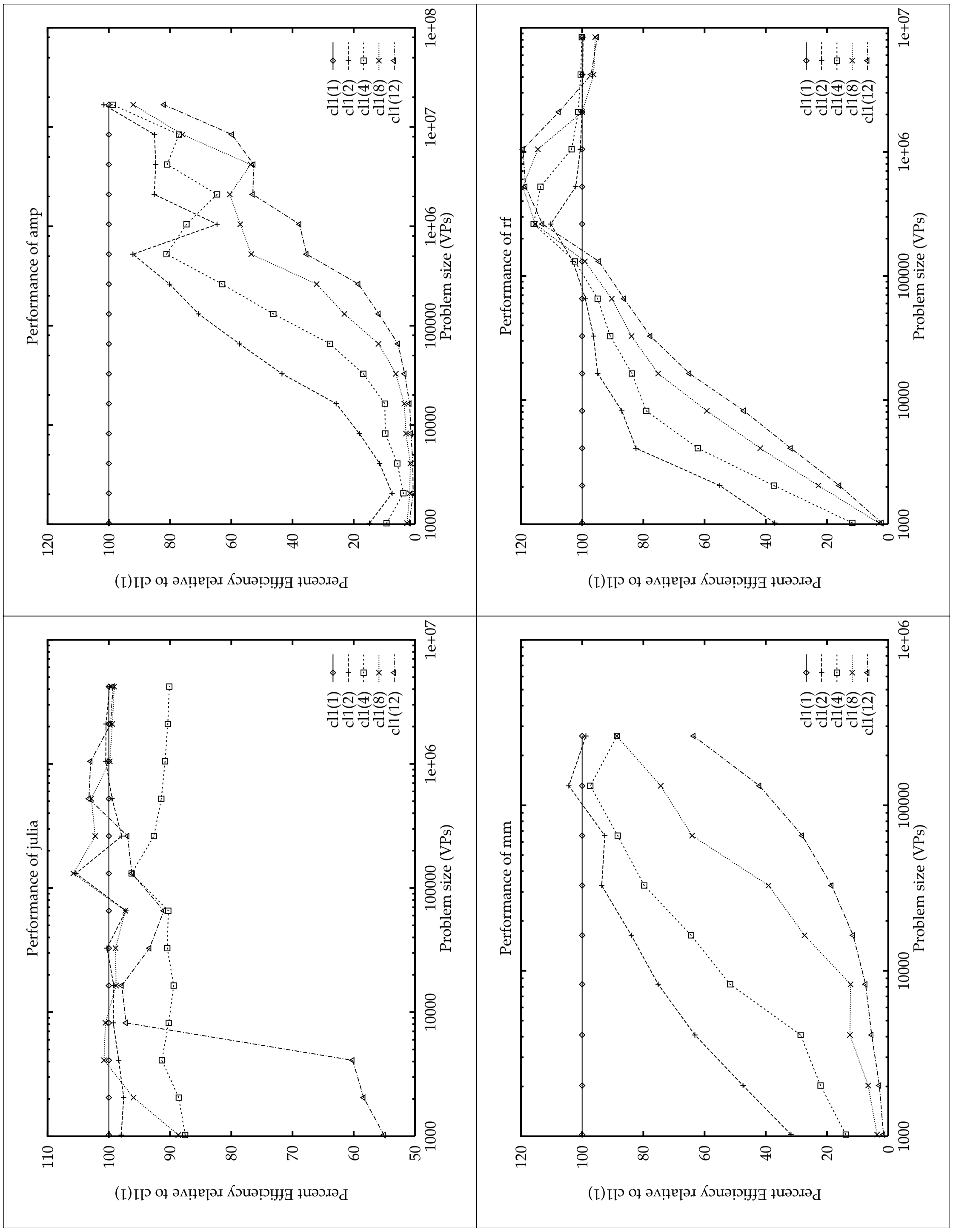


Figure 7.3b: Cluster Efficiency (Relative to pC*–1 processor) (Part 2)

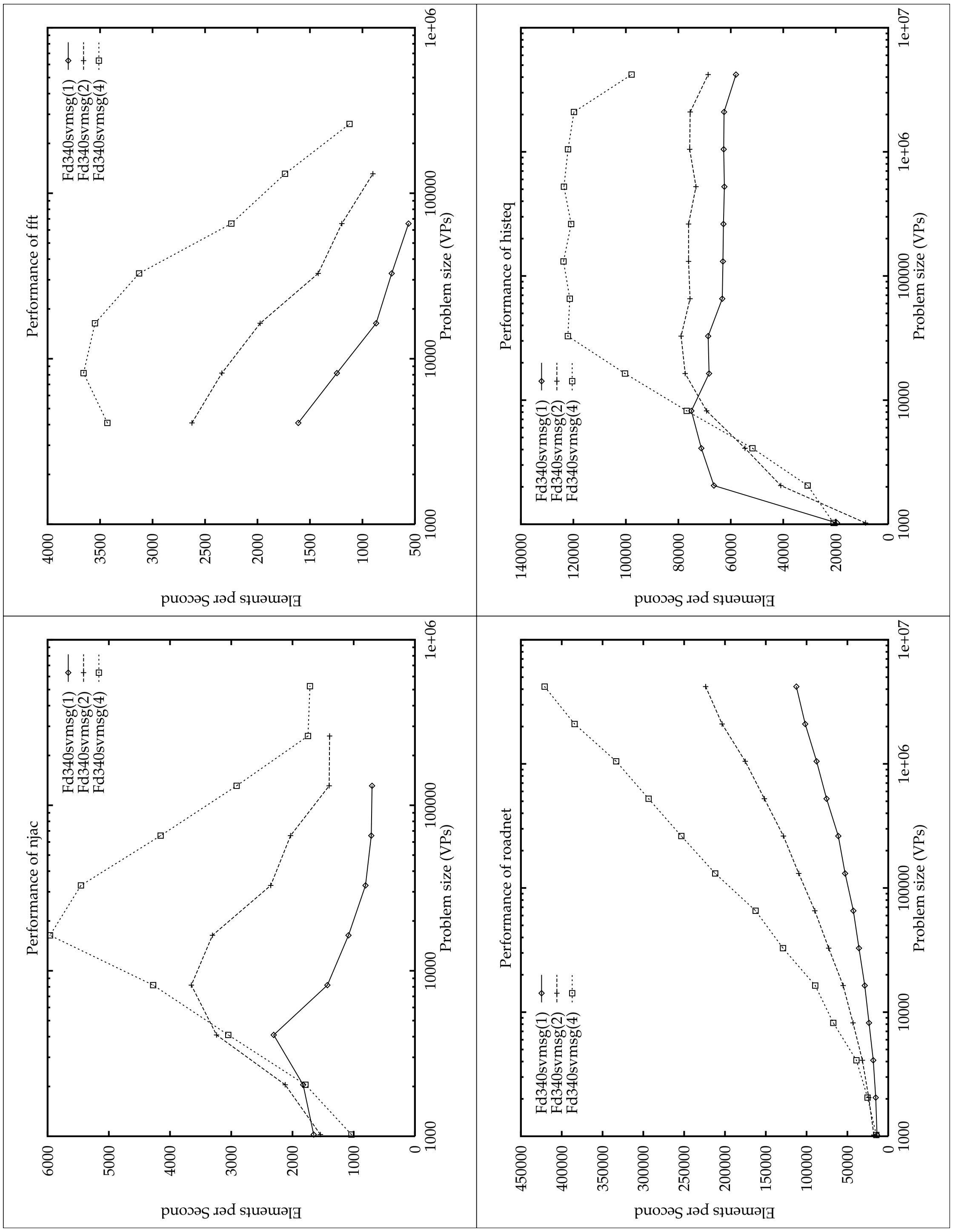


Figure 7.4a: SGI Performance (Part 1)

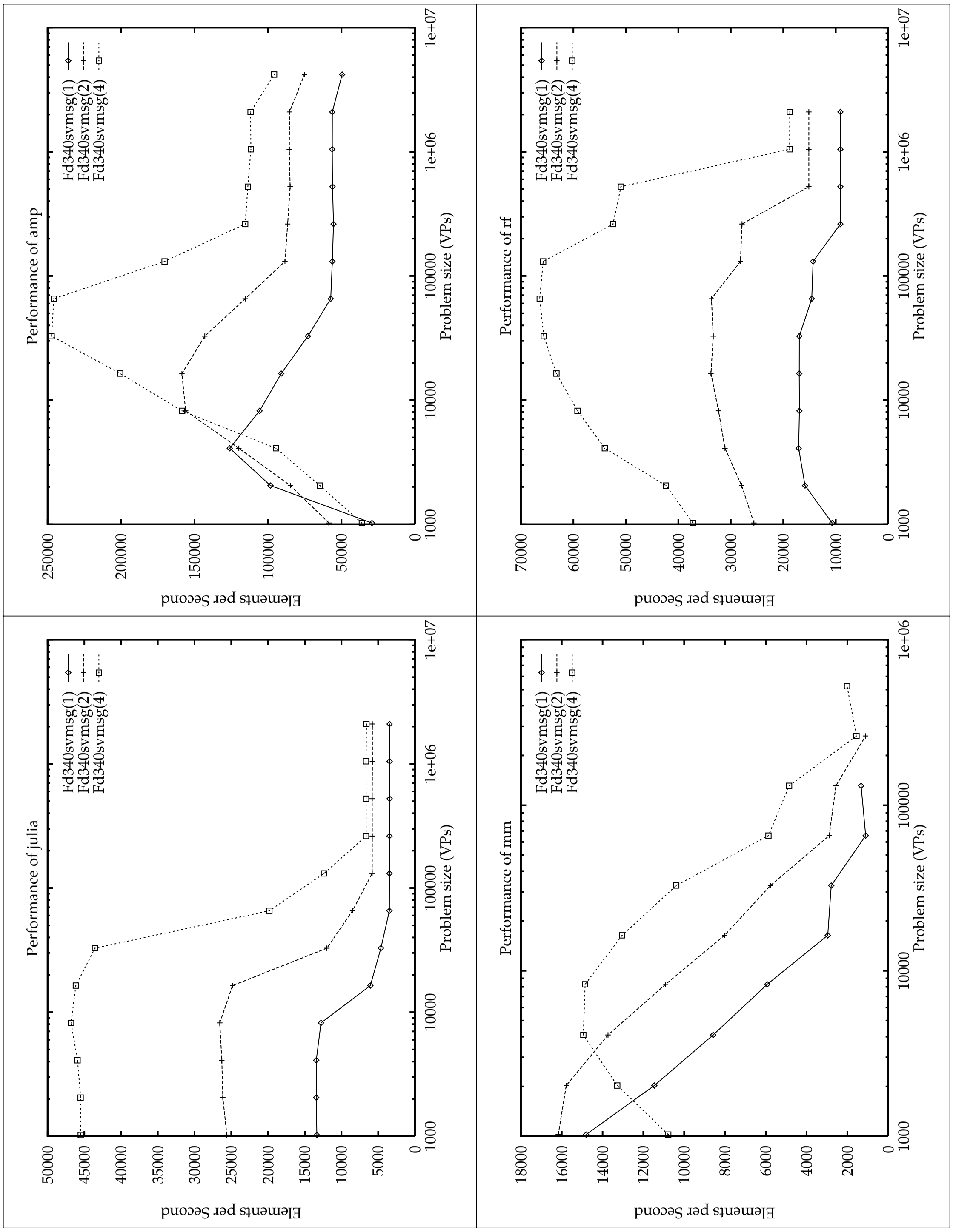


Figure 7.4b: SGI Performance (Part 2)

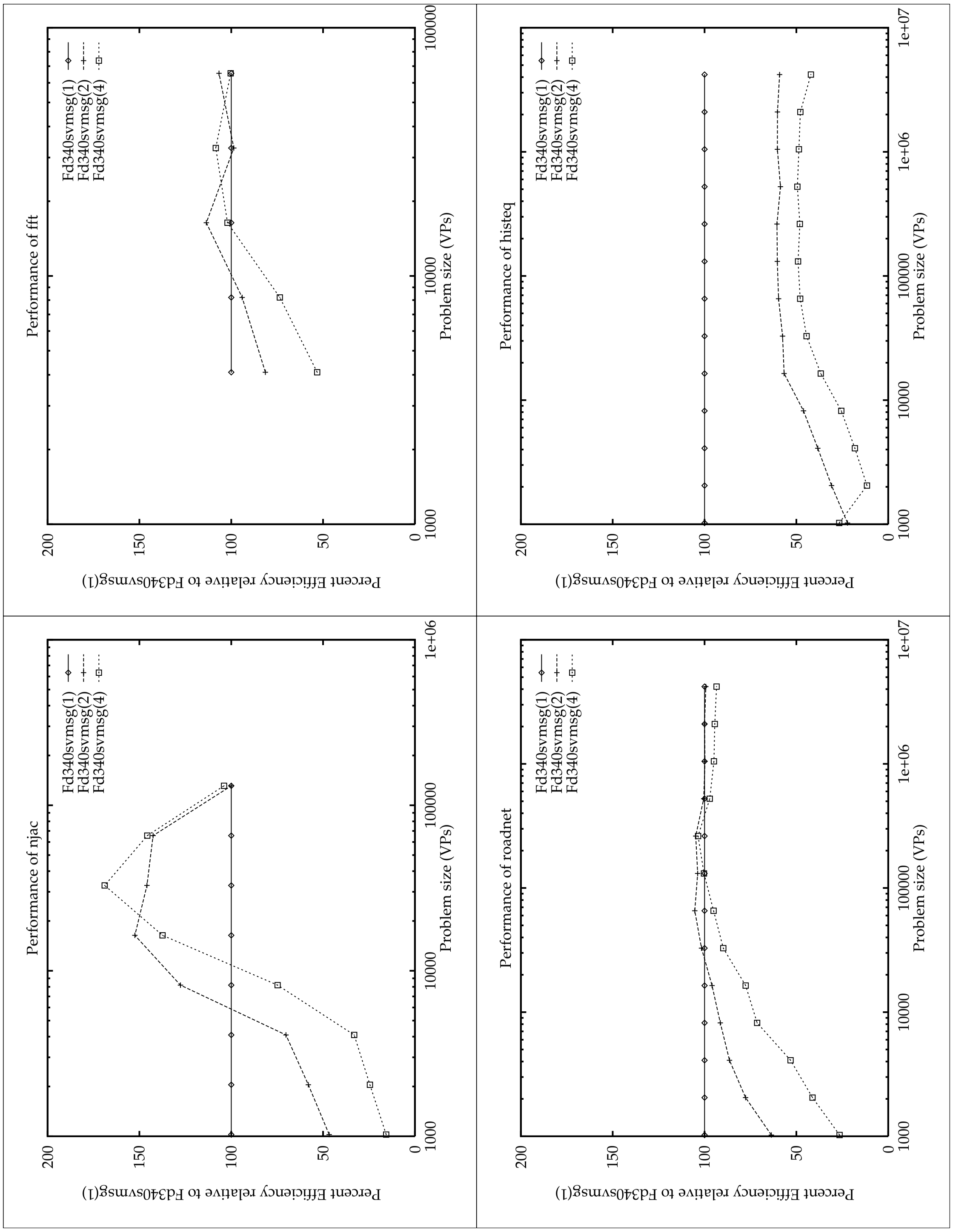


Figure 7.5a: SGI Efficiency (Relative to pC*–1 processor) (Part 1)

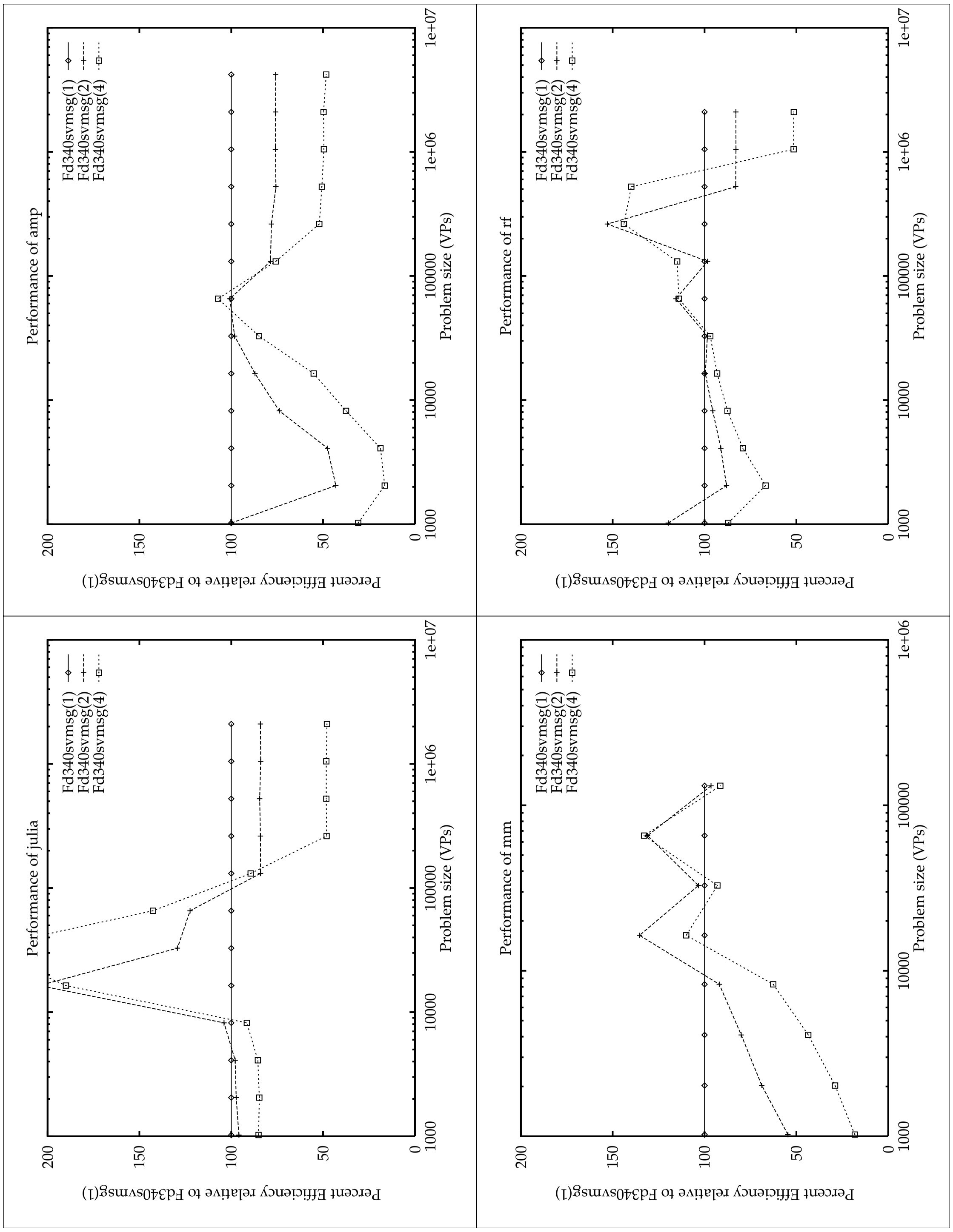


Figure 7.5b: SGI Efficiency (Relative to pC*–1 processor) (Part 2)

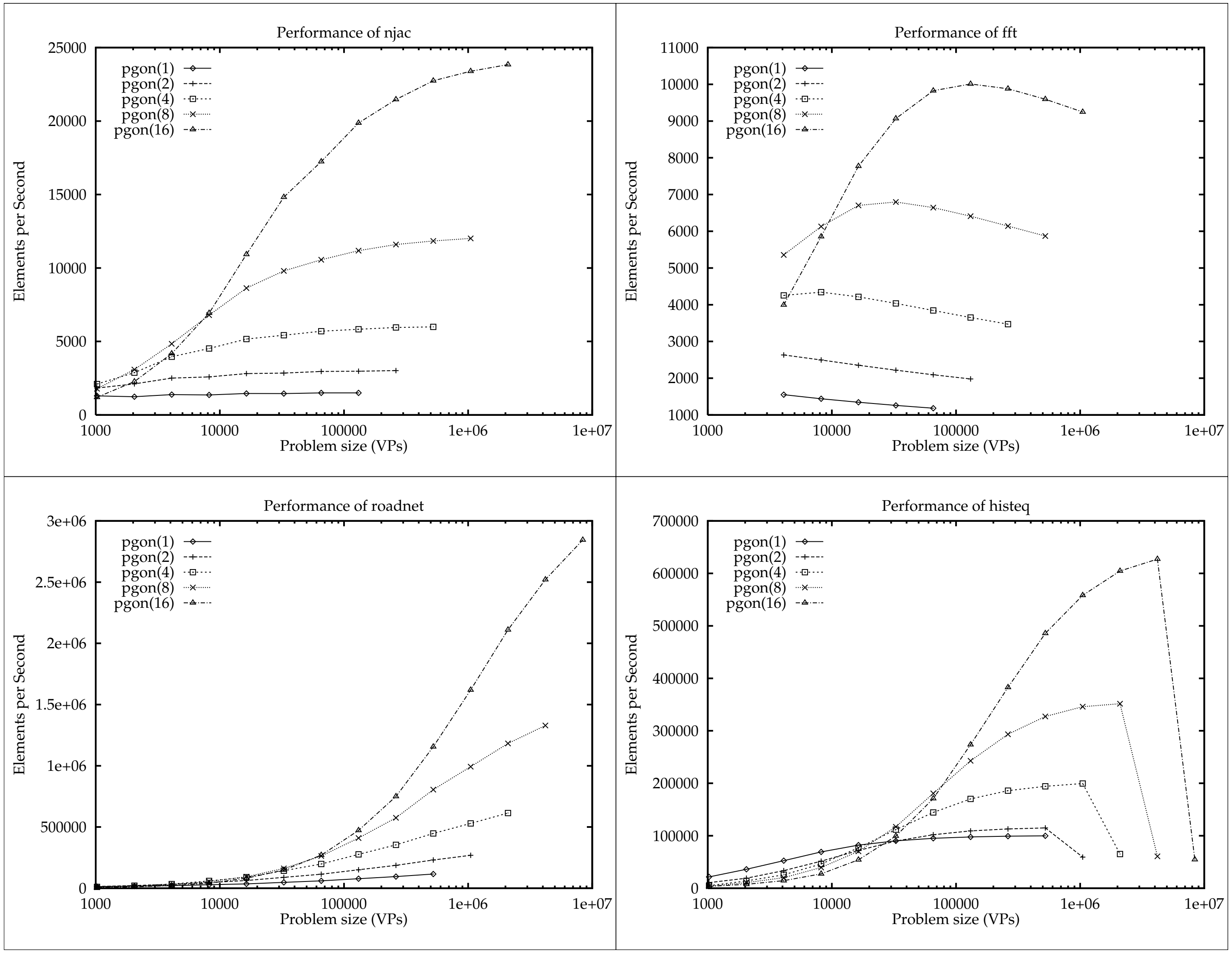


Figure 7.6a: Paragon Performance (Part 1)

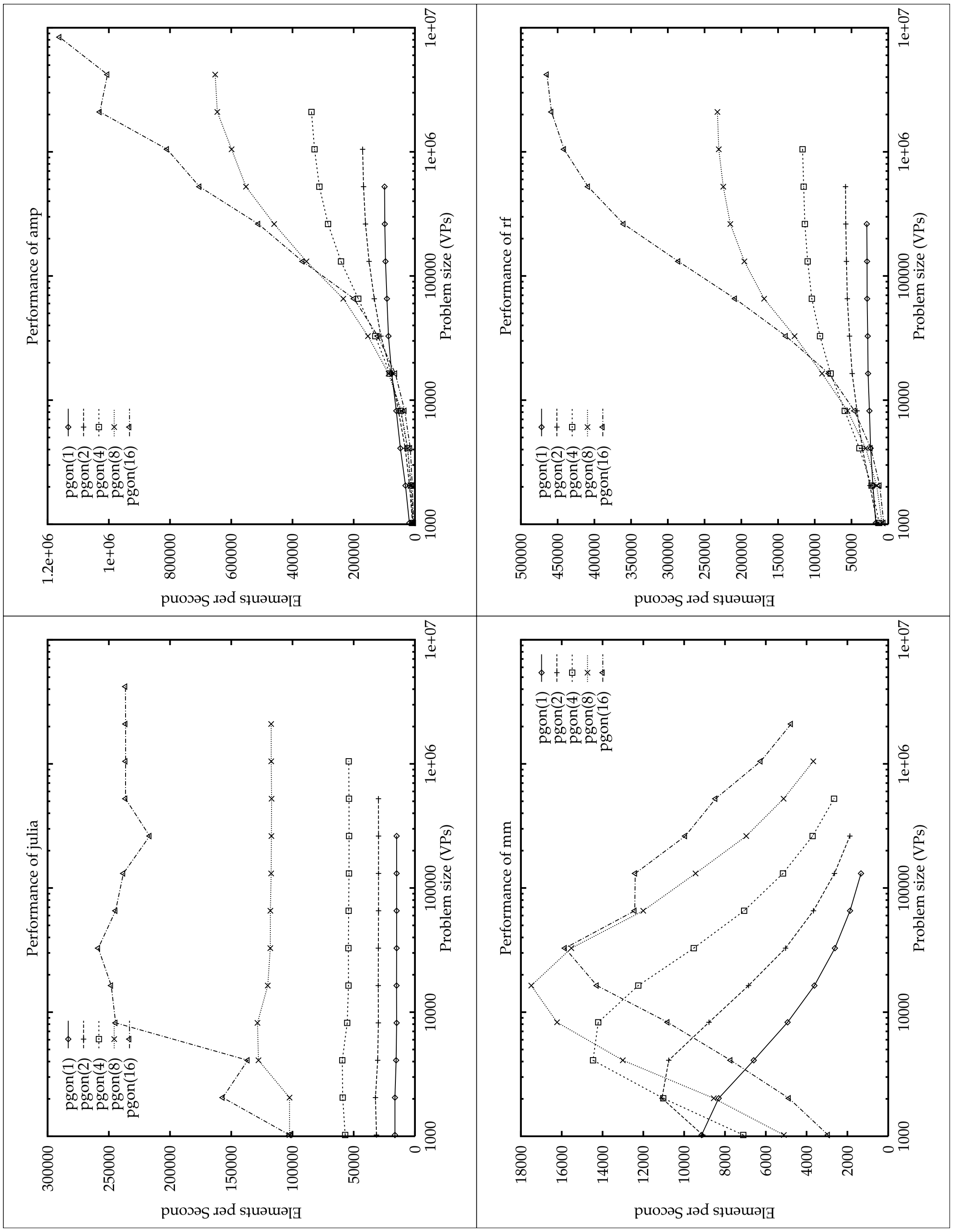


Figure 7.6b: Paragon Performance (Part 2)

The efficiency results for the Paragon do not exhibit any surprises (figure 7.7). They tend to be higher than corresponding cluster efficiencies for problems that are dependent on network performance, such as fft and njac. Relative performance figures comparing the cluster with the Paragon when equal numbers of processors are used in each are shown in figure 7.8. It is interesting to note that the 50MHz i860 chips in the Paragon result in performance which is roughly two to four times slower than the 60MHz Sun systems, with Paragon performance best on the most communications-intensive benchmark, fft, where 10Mbps Ethernet cannot compete with the Paragon's low-latency 30MBps (240Mbps) mesh system.

7.6 Performance of pC* Contrasted with Sequential C

All previous results have been intra-system: i.e., they compared pC* with itself on different architectures or cluster sizes. To gain a proper understanding of the absolute performance of the system, it is necessary to compare it to independently developed systems running, where possible, on the same hardware.

For the first comparison we will consider how the C* benchmarks described in section 7.2 compare with sequential ANSI C solutions of the same problems. In all cases, the C implementation was one involving at least some thought; all obvious, and a few unobvious, algorithmic optimizations were performed. In particular, the C implementations differ in the following ways from the C* ones:

fft — In the most major difference, the Fast Fourier Transform code used was taken from the netlib fftpack version 4, written in Fortran by Paul Swarztrauber at the National Center for Atmospheric Research.[5] It was translated to C using f2c 19950110, a Fortran-to-C translator from AT&T Bell Labs.[6] The interface to these FFT routines uses a pre-processing step which is shared by both the forward and inverse transformations, amortizing cost between the two phases in a way that is not done with the C* butterfly implementation. The C implementation is therefore highly optimized, and the C* version will suffer accordingly when compared with it; nonetheless, if a Fourier algorithm is required in a C program, re-use of packaged code like this is the appropriate choice.

histeq — The histogram equalization implementation can simply use the pixel values as indices into an array, whose elements are incremented immediately; the overhead is significantly less than the C* invocation of general send.

njac — The Jacobi iteration implementation benefits from not requiring four temporary values that are the size of the array when computing the stencil.

5. Available via ftp from `netlib.att.com:netlib/fftpack`.
6. Available via ftp from `netlib.att.com:netlib/f2c`.

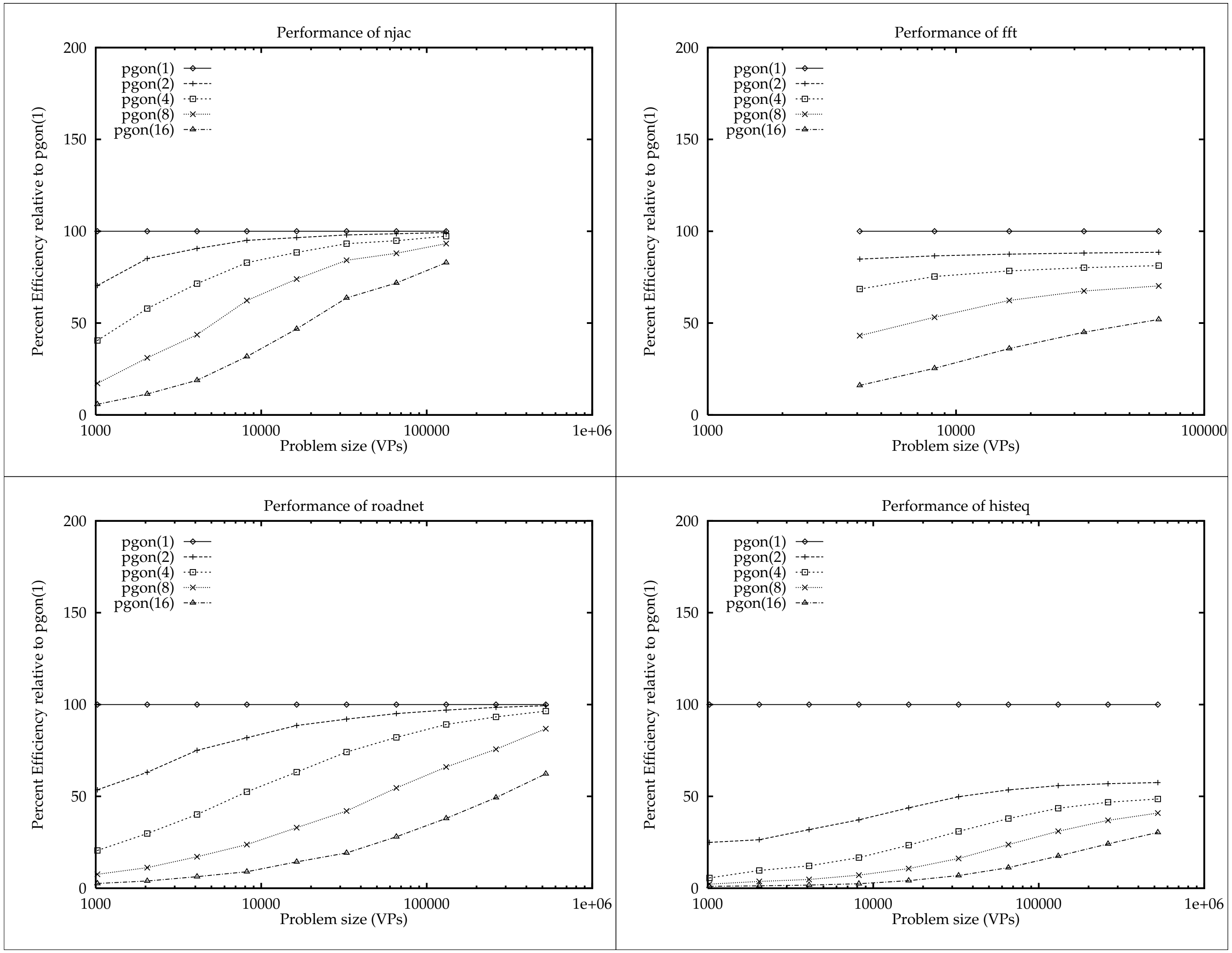


Figure 7.7a: Paragon Efficiency (Relative to pC*–1 processor) (Part 1)

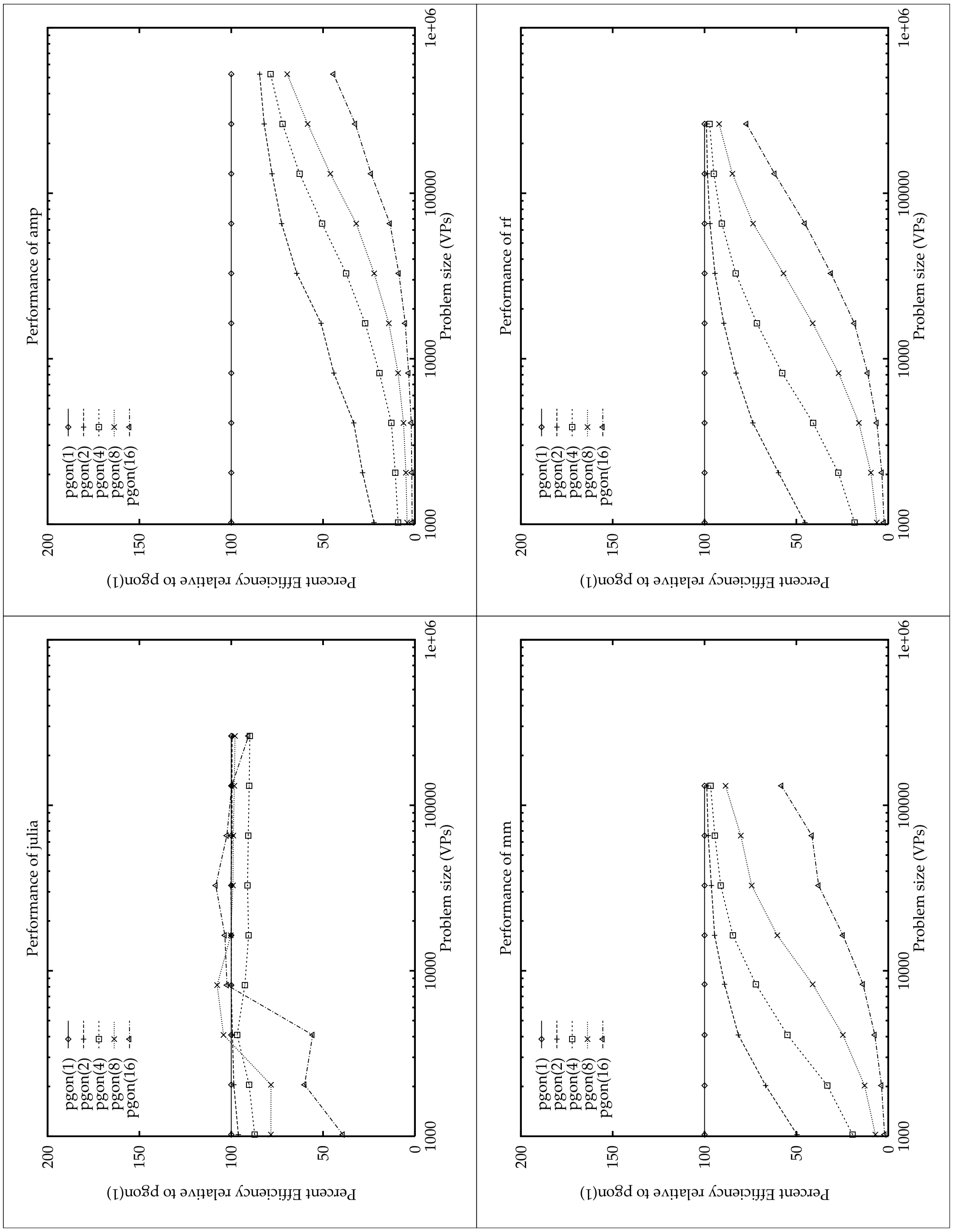


Figure 7.7b: Paragon Efficiency (Relative to pC*–1 processor) (Part 2)

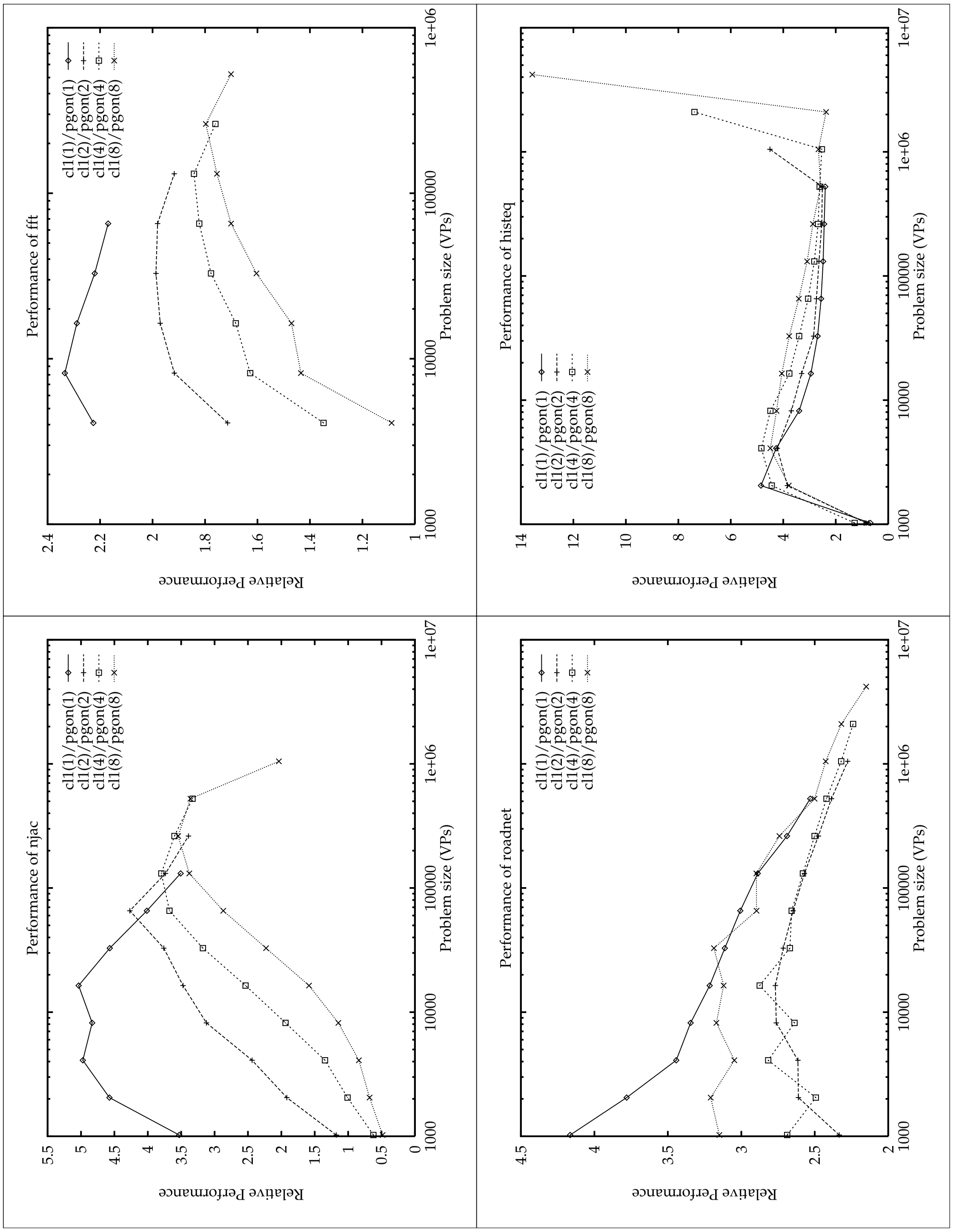


Figure 7.8a: Relative Performance Cluster / Paragon (Part 1)

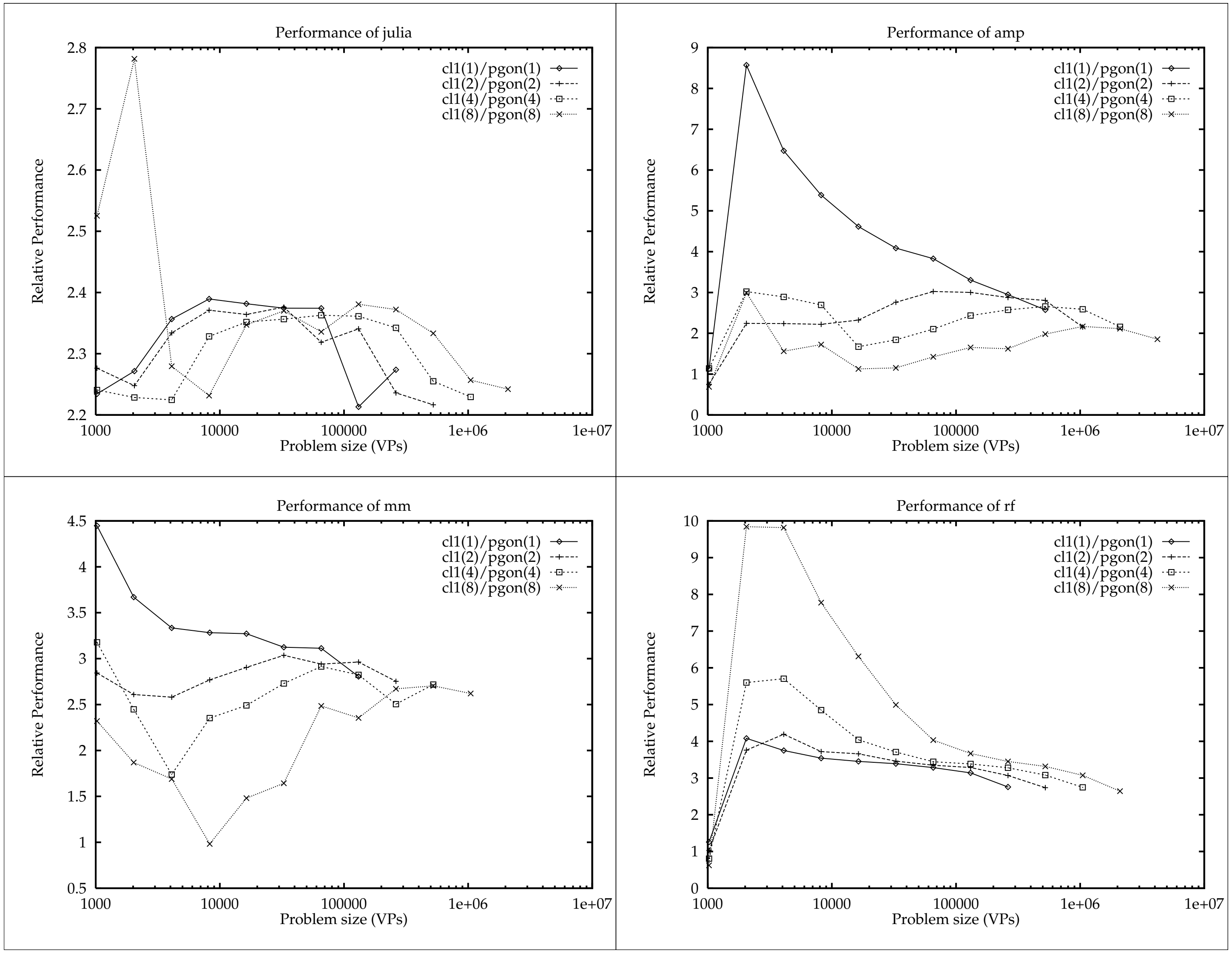


Figure 7.8b: Relative Performance Cluster / Paragon (Part 2)

roadnet The C implementation of the distance-to-road problem maintains a list which contains exactly the points which are on the current road perimeter, sorted by x and y coordinates (in an array, and linked list per array bin, respectively, to permit fast searches). Therefore it, like the C* implementation, gains from operating only on positions of interest, but does not have to examine the entire map to determine the current perimeter.

amp The amplitude screener is a reasonable implementation, performing a scan of all elements in the window around each pixel in turn to compute the average. Depending on the window size tested, a prefix-scan implementation similar to the internals of the C* version could decrease the computation cost, by maintaining a running sum and subtracting the value from one side of the window when adding the value from other side. We chose not to implement this more complicated algorithm for benchmark purposes alone, since we use a constant window size (3×3) in all tests.

julia There is no essential difference between the C* and C implementations.

mm The C version uses a straightforward triply-nested loop with accumulation of vector dot products into a temporary value.

rf The region around each pixel is scanned, and a bubble-sorted list of neighbors yields the window median. The implementation is much more cache-friendly than the C* one, which has the same general algorithm but, due to data-parallel treatment of the image, puts the "loops" over rows and columns within the sorting code.

The speedup of the cluster relative to the C implementation running on a single cluster node is shown in figure 7.9. The trends are essentially the same as those for speedup relative to pC* on one cluster node, except for roadnet where the C version improves faster with larger problem sizes than the C* version. In three benchmarks—fft, histeq, and roadnet—the C* version does not succeed at outperforming the C version, even with twelve processors. However, all three do approach to nearly half the sequential performance, with fft continuing to improve as problem size increases.

The other benchmarks perform better in comparison to C on the cluster. julia is best, outperforming C with two processors, and continuing to improve with an efficiency of about 80%. On larger problem sizes, matrix multiplication is faster with two processors, while the remaining three benchmarks require four processors to outperform the sequential implementations.

On the SGI (figure 7.10), performance is less impressive, due for the most part to the fact that the C implementations uniformly require less memory than the C* ones, so relative performance suffers as the C* versions quickly result in bad cache behavior. julia performs quite well with 80%–90% efficiency until data sizes exceed the cache, and rf has a respectable 50% efficiency on the smaller problem sizes. The most interesting result is for matrix multiply, where the C* version uses more memory but accesses it in a more cache-friendly manner, resulting in efficiencies of 60%–100% for the larger data sizes.

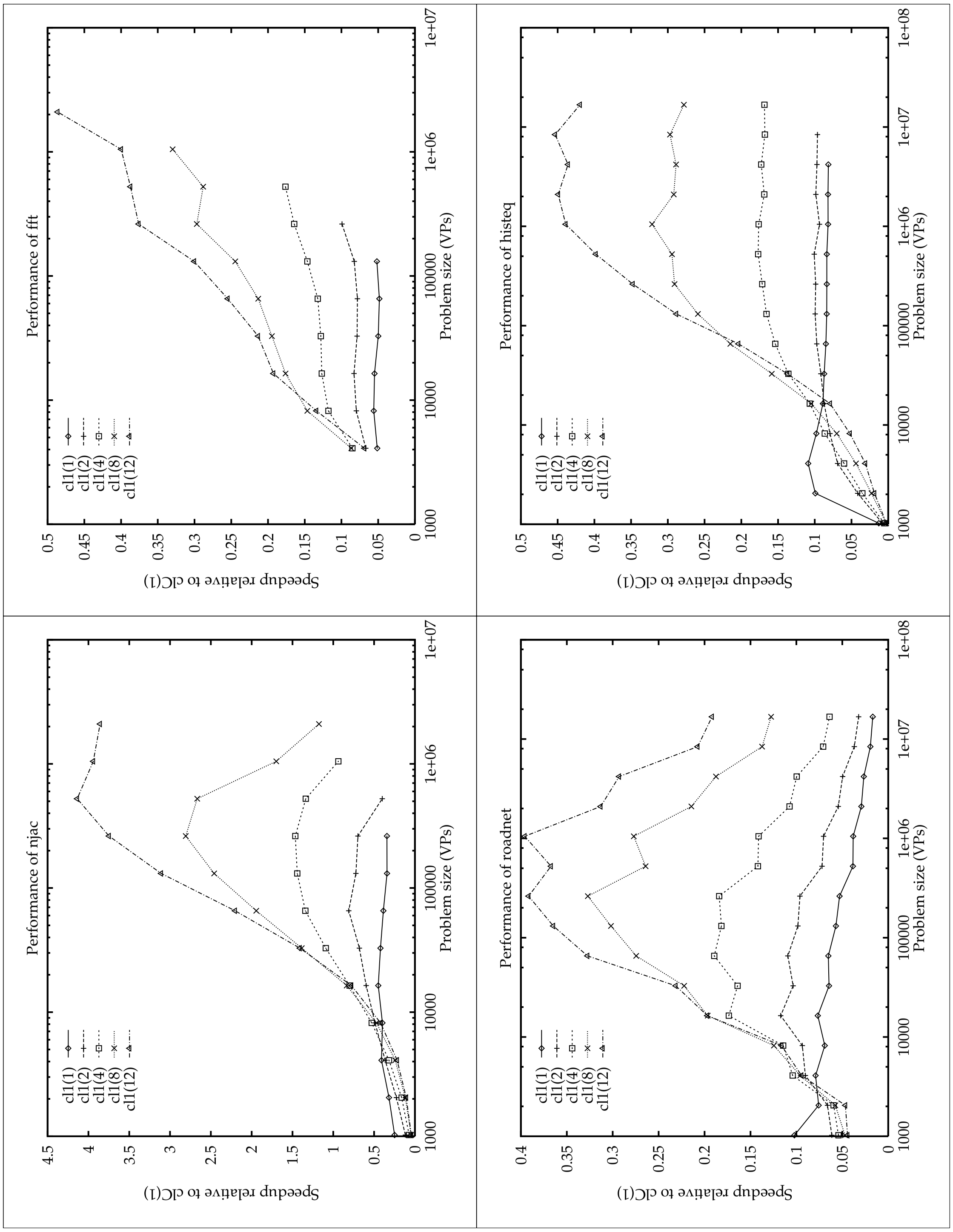


Figure 7.9a: Speedup of Cluster pC* Relative to C (Part 1)

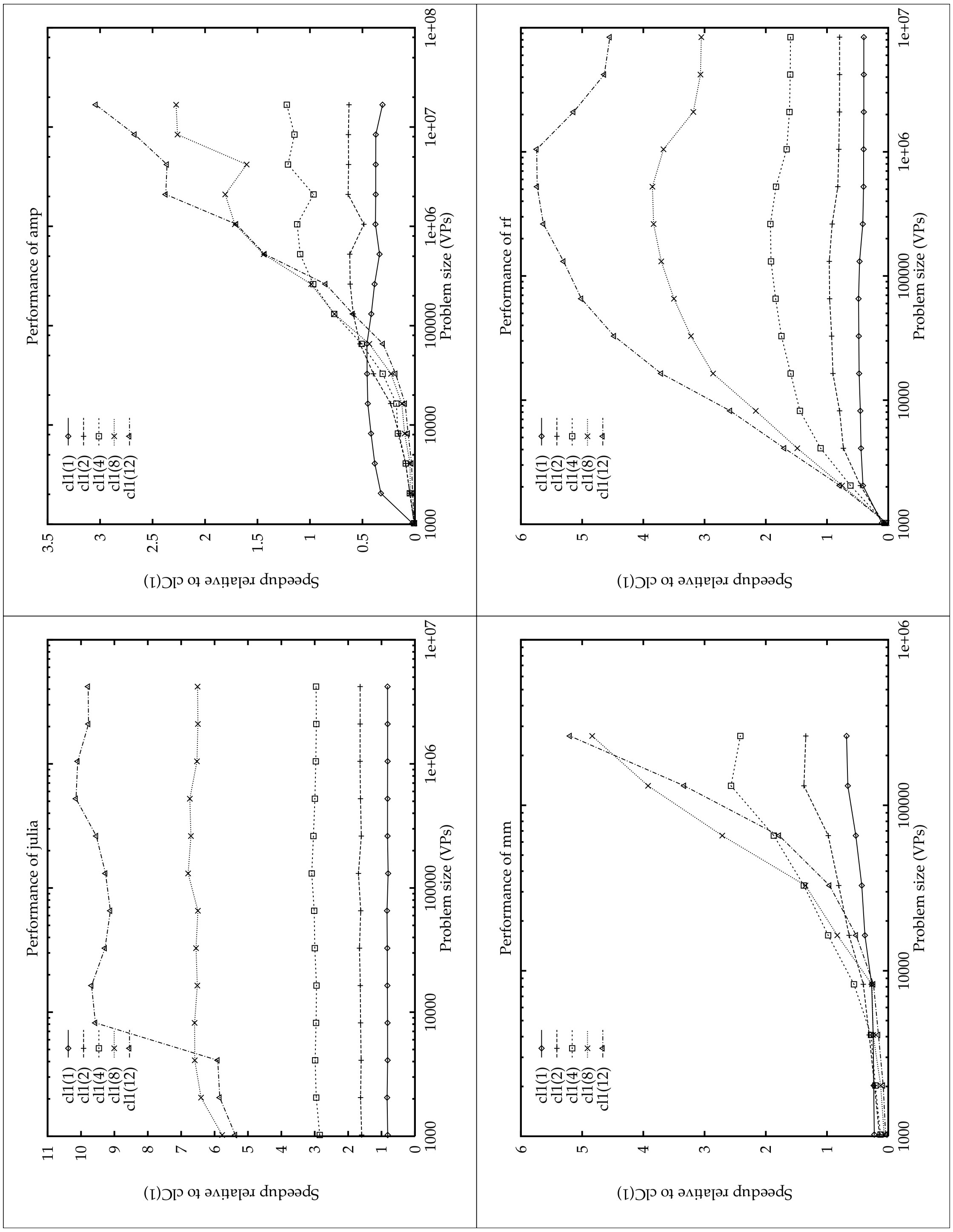


Figure 7.9b: Speedup of Cluster pC* Relative to C (Part 2)

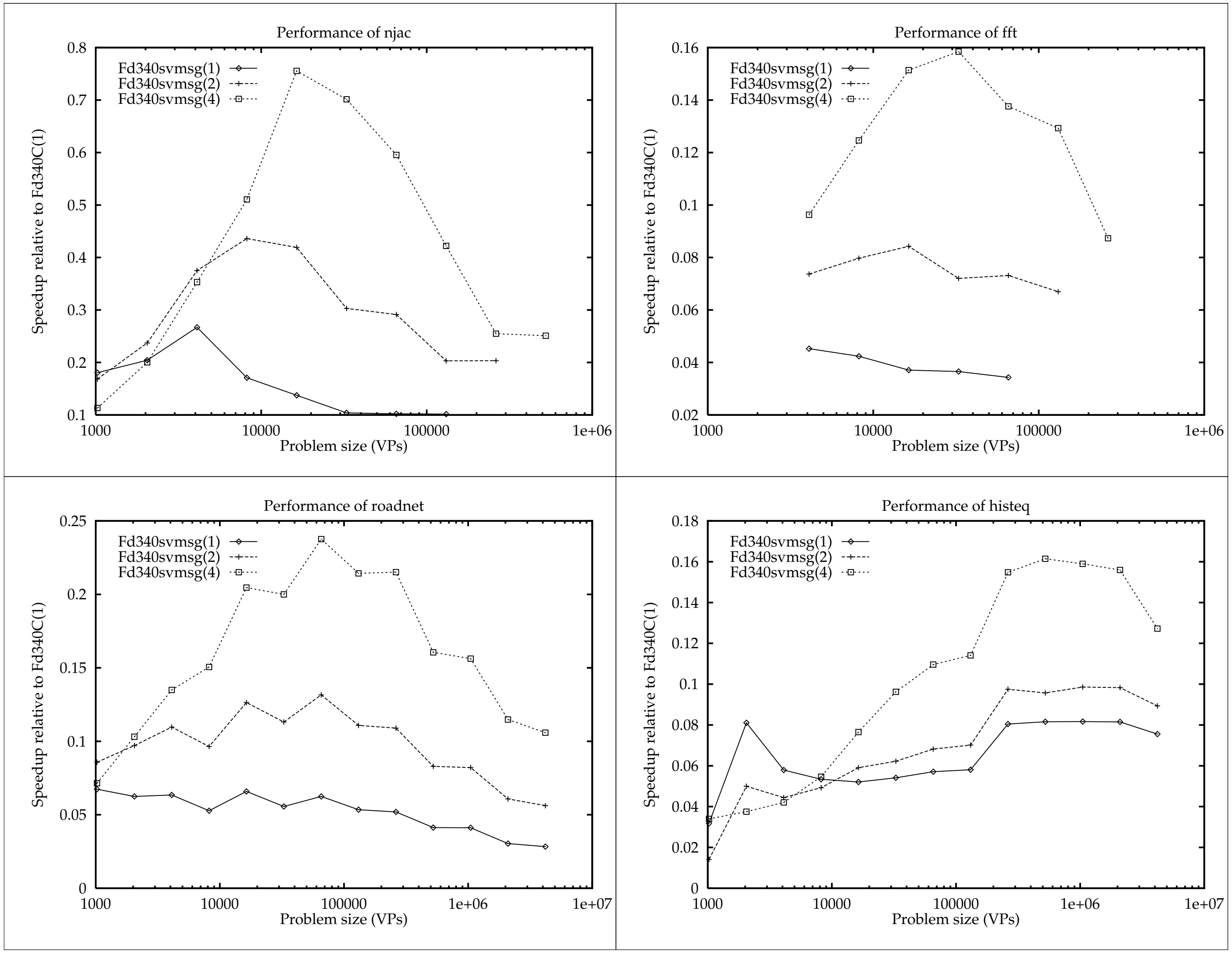


Figure 7.10a: Speedup of SGI pC* Relative to C (Part 1)

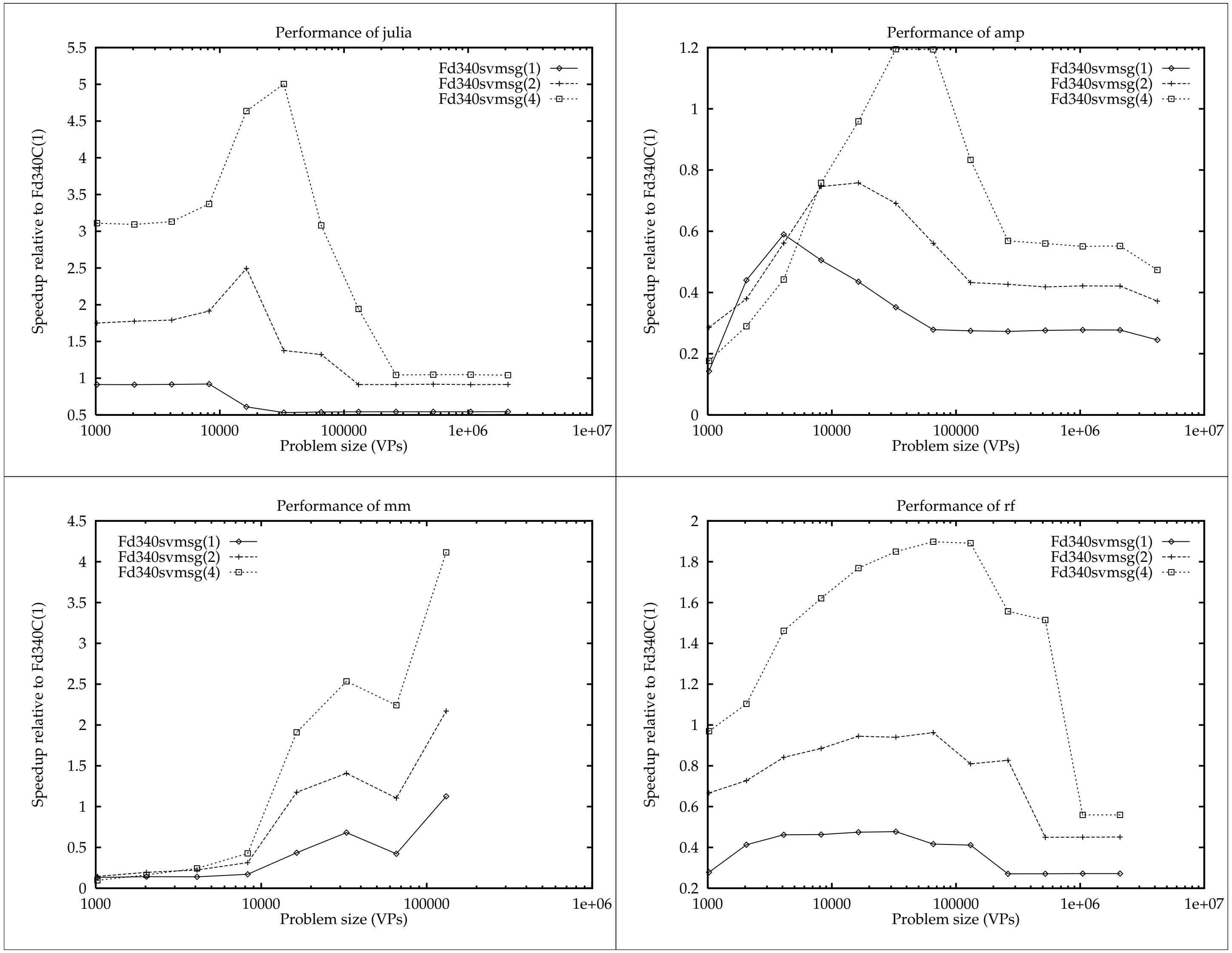


Figure 7.10b: Speedup of SGI pC* Relative to C (Part 2)

7.7 Performance of pC* Contrasted with TMC C*

The next comparison is to run the C* benchmarks on an architecture with a highly optimized special vector processor and fast network, and a compiler designed specifically for that hardware: to wit, TMC C* 7.2 on a CM5 supercomputer. Raw performance numbers are given in figure 7.11. The efficiency of the TMC implementation on its native architecture is impressive, with all benchmarks except the communications-intensive histeq having efficiencies in excess of 80%, and often in the high 90%s. The CM5 implementation does not do the sort of communication collision detection that we describe in chapter 5, so performance drops considerably on histeq due to the bottleneck of data transfer between nodes.[7]

A very interesting performance issue, first raised in chapter 3, can be seen in the behavior of matrix multiply, where to increase the data size by (roughly) doubling at each step while retaining a square shape, we were obliged to specify shapes that did not have a power-of-two extent along each axis. Shapes with power-of-two sides were perfectly allocated on the CM5 with square subgrids stored on each node, but this could not be done for the oddly sized shapes. The CM2 SIMD implementation of C* padded shape sides to powers-of-two; the CM5 is not quite so restrictive, but still requires decomposing the vector units into a physical grid whose sides are powers of two; the subgrid assigned to each vector unit must be the same; and the subgrid size must be a multiple of 8 (see Appendix B of (Thinking Machines Corporation, 1993) for details). The effect of these restrictions is to cause elongated node subgrids and wasted space. For example, on the 256 node (1024 vector unit) CM5, the shape which is 724×724 is allocated as 1024 subgrids of size 92×6, causing an overallocation of 7.8%, as well as a fifteen-to-one imbalance in the length/width ratio of the nodal subgrids. It is to be expected that, had we not chosen powers-of-two for the input data sizes on the other benchmarks, similar undesirable behavior would have been exhibited on them as well. The ability of pC* to partition the shape into subgrids which need not be the same on all nodes is a significant benefit.

The speedup of the cluster relative to the 64 node CM5 is shown in figure 7.12. For the most part, the results are as expected when comparing a multi-million dollar 64-processor supercomputer with a set of twelve workstations connected by Ethernet. However, it is very heartening to notice that the cluster outperforms the CM5 in absolute performance on three of the eight benchmarks: histeq, due to the general communications optimizations in chapter 5, will be faster with eight nodes; roadnet, with the context optimizations described in chapter 3, will be faster with as few as four nodes; and rf, due to the efficient grid communications routines of chapter 6 and use of the context inlining mentioned in chapter 3, will be faster with eight to twelve nodes, depending on data size.

In terms of raw compute power, the CM5 should be three (64 32MHz SPARCs compared with 12 60MHz SPARCs) to twelve times faster than the cluster, depending on how well the benchmarks take advantage of the four-processor vector units available on each

7. The library implementation of general get in TMC C* has an additional parameter that specifies the expected collision behavior of the operation, but the description implies this is mostly intended to decrease memory requirements on large shapes. There is no documented analog for colliding sends.

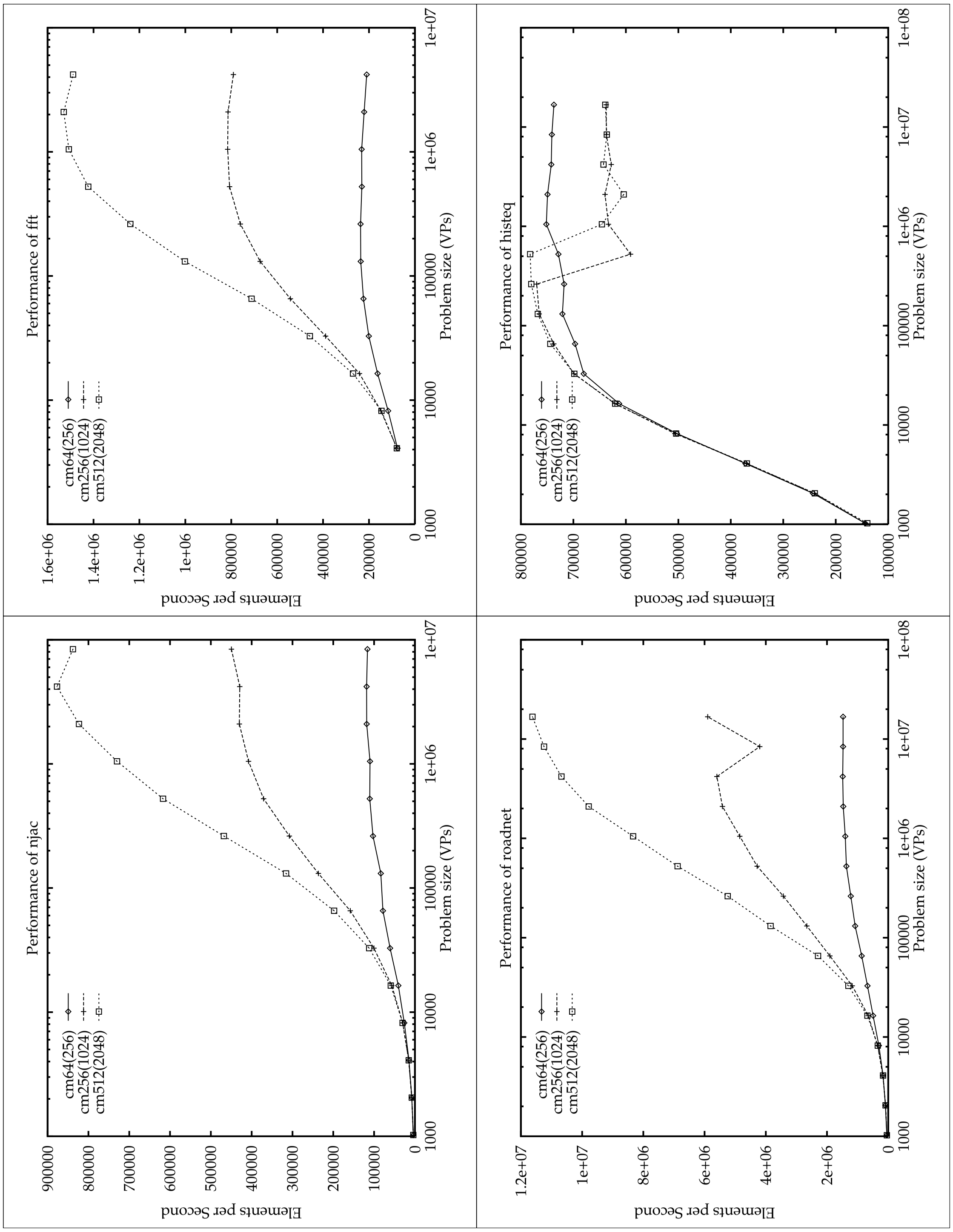


Figure 7.11a: Performance of Benchmarks on CM5 (Part 1)

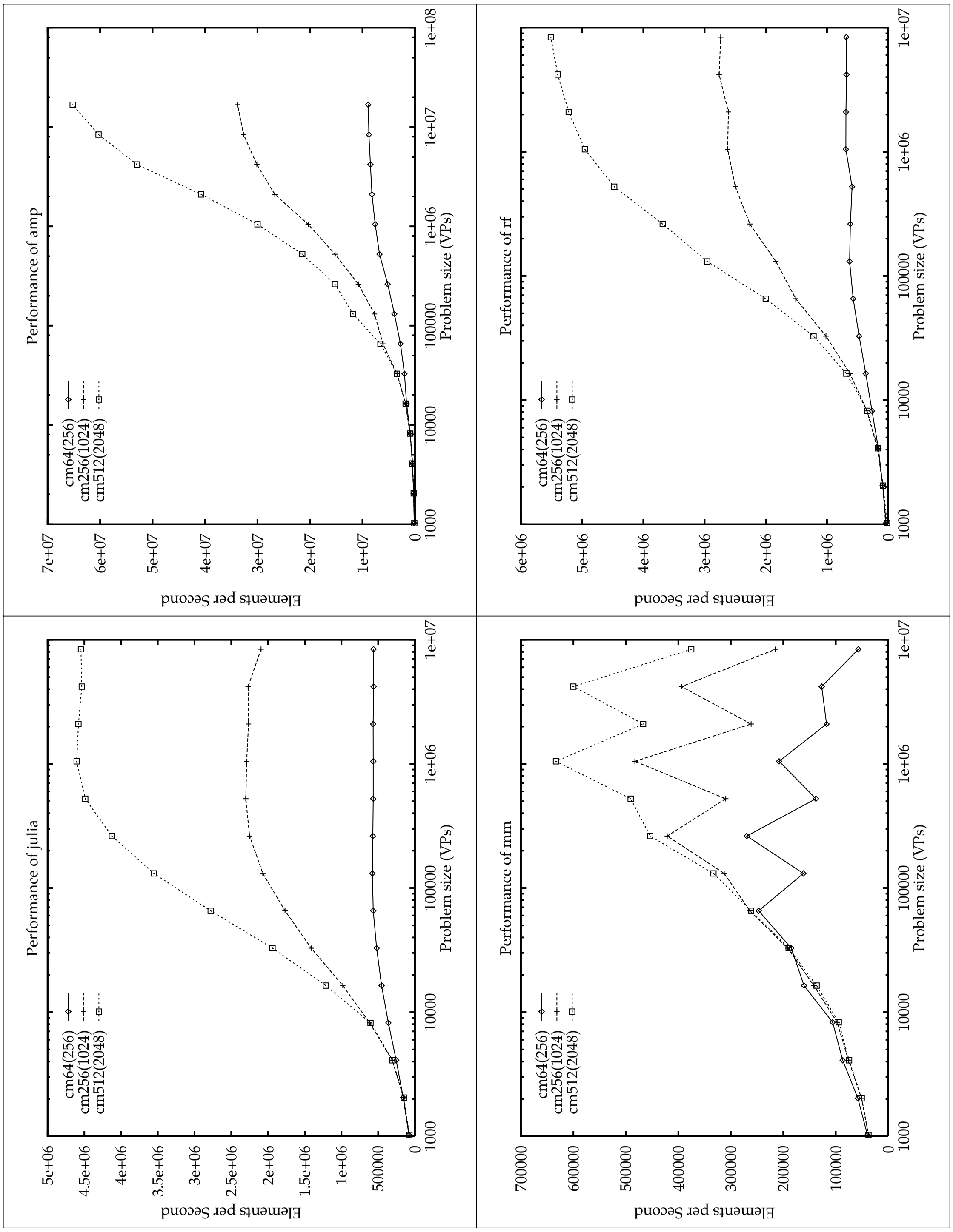


Figure 7.11b: Performance of Benchmarks on CM5 (Part 2)

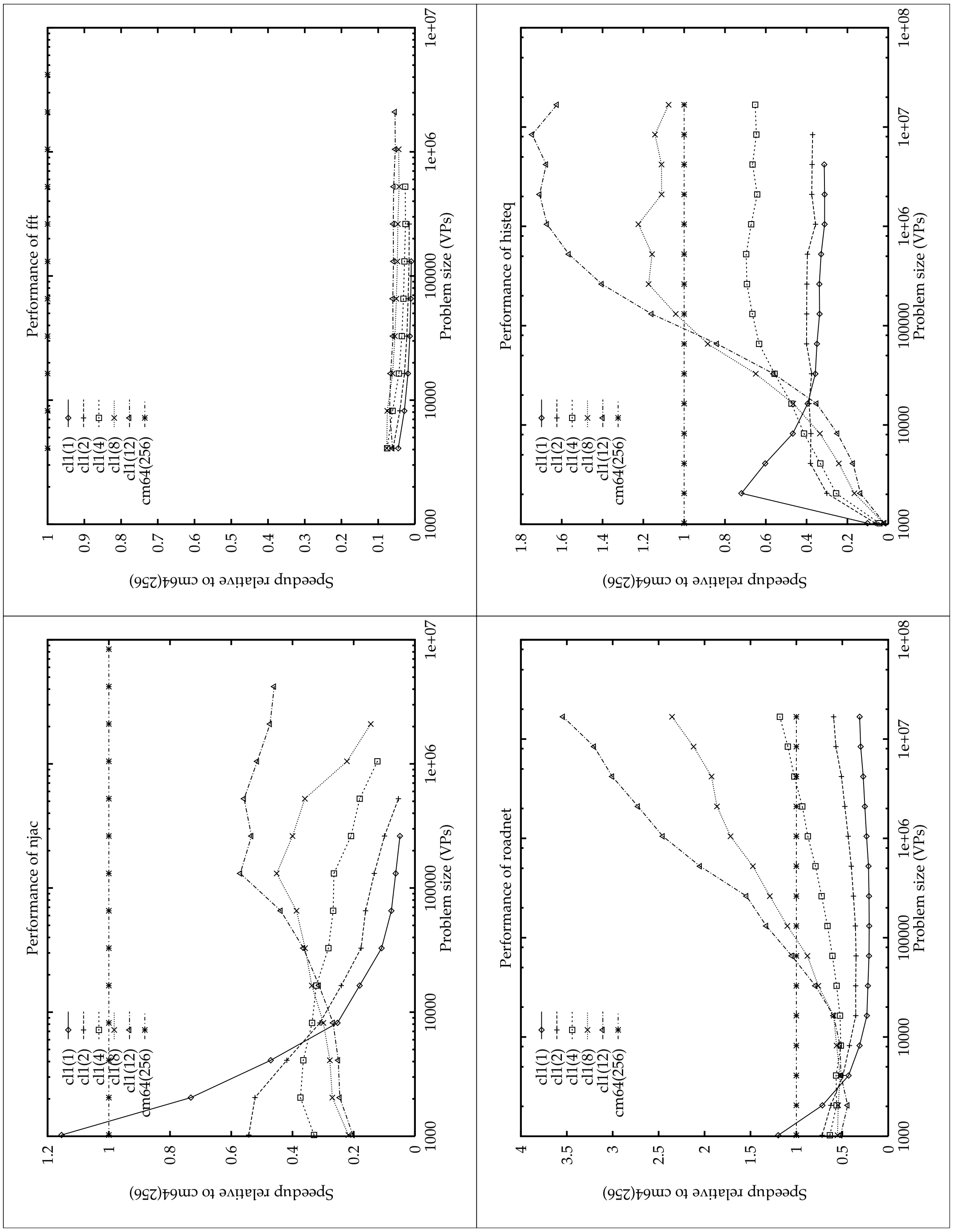


Figure 7.12a: Speedup of Cluster Relative to CM5-64 (Part 1)

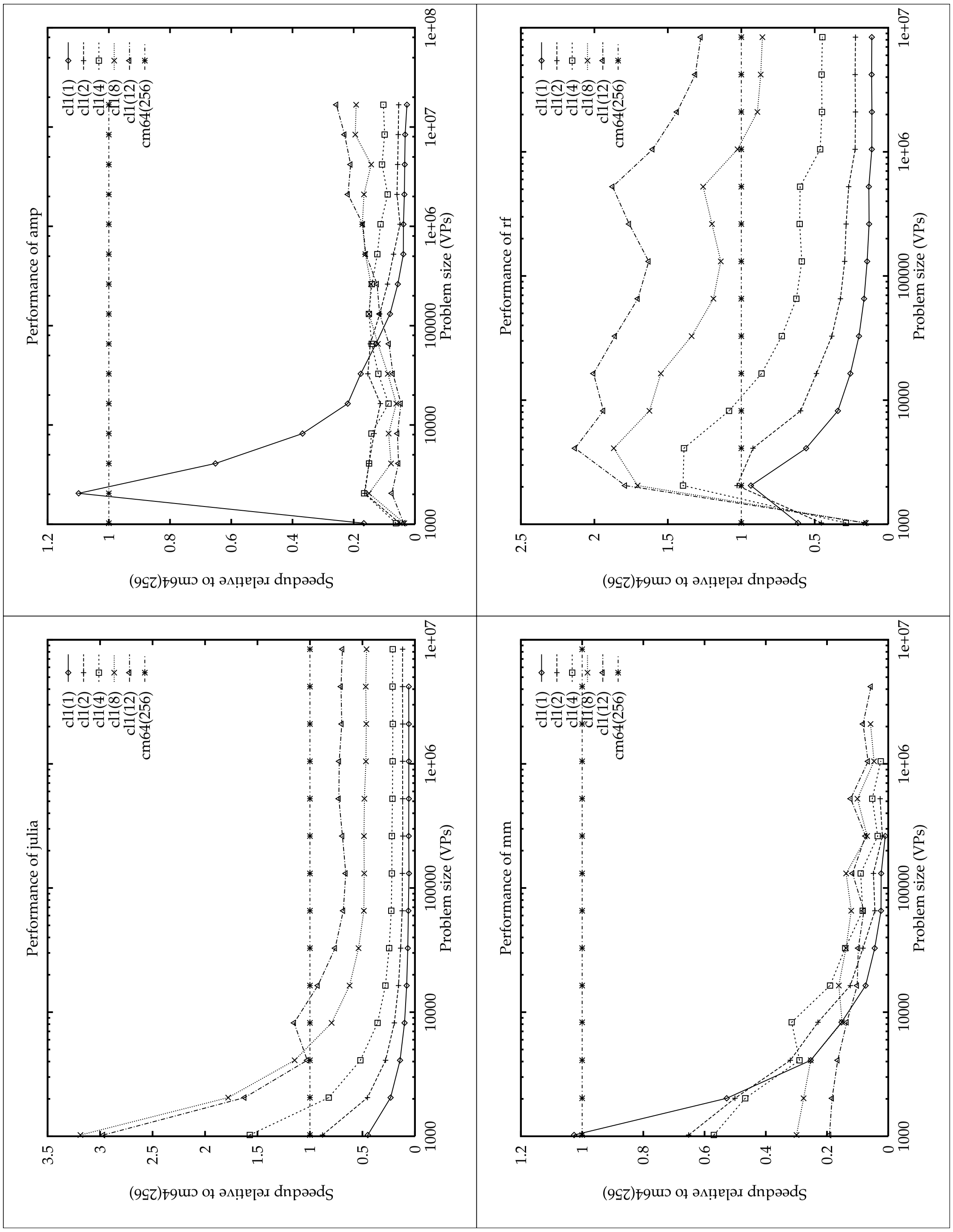


Figure 7.12b: Speedup of Cluster Relative to CM5-64 (Part 2)

node. In fact, its advantage, when scaled to per-processor performance, is less impressive, as shown in figure 7.13. Only on fft does the CM5 clearly outperform the cluster, due no doubt to its high speed interconnect. It also outperforms the cluster on the larger mm benchmarks, though not to such a marked degree. On the rest of the benchmarks, per-processor performance seems to be between one half and one eighth that of the cluster. Note that, although (for example) the 64 node CM5 has 256 vector unit processors, we are taking the conservative view that these processors are not effective for the benchmarks considered, and instead assume each CM5 has as many processors as it has Sparc nodes. The per-processor advantage of the cluster is multiplied by four if the alternative interpretation is taken.

7.8 Performance of pC* Contrasted with UNH C*

The final comparison is between pC* and the most recent version of its parent system, the UNH C* compiler from the University of New Hampshire.[8] To permit a reasonable comparison between the systems, we use the PVM inter-process mechanism for each, and modified the UNH job startup mechanism to match that of pC*.[9] We were able to test only three of the eight benchmarks—fft, njac, and roadnet. julia encountered a bug in the UNH translator, and the remaining programs rely on auxiliary routines that are not available in the UNH runtime library.

The relative performance of the two systems is shown in figure 7.14. The performance of fft indicates that the communications method described in chapters 4 and 5 is roughly two times faster than the mechanism in (Lapadula & Herold, 1994). The latter algorithm falls down in particular when the cluster size is not a power of two (cf. the performance curve for twelve nodes).

On njac, pC* outperforms UNH by approximately a factor of three, except at higher data sizes on clusters with 2^k nodes where the performance drops to a factor of 2, and at small data sizes with large clusters where the UNH implementation is faster. Degradation of pC* in both these cases is due to the performance problems with `reduce` described in section 7.3. The UNH system implements an algorithm equivalent to our LOGLOCEX of section 4.4.2, which uses half the exchange steps that pC*'s algorithm does and does not suffer as badly on power-of-two meshes. Comparing njac to a version without the reduction operation indicates that, on a 2M-element problem with eight nodes, 10% of the UNH runtime goes to reductions, compared with about 21% of the pC* runtime. On small problems, pC* per-

8. We used UNH C*, version 950609 from June 1995, available from `ftp.cs.unh.edu:pub/cstar`. The runtime library was compiled with the same flags as that of pC*: `-O2 -DNDEBUG -msupersparc`, using gcc 2.6.3. C* programs were translated using flags `-O2 -DNDEBUG -msupersparc -debug=308`, the latter being recommended to us by Phil Hatcher as turning on the largest number of useful compile-time optimizations, such as schedule-based communications (Mason *et al.*, 1994).

9. Initial tests with PVM indicated that `pvm_spawn`, when permitted to assign processes to machines under its own rules, did not provide a sufficiently fine control over what machine executed worker jobs, sometimes putting two workers on the same machine while an available machine was left idle. The pC* and UNH implementations described here assign jobs to all machines listed in the PVM configuration in a round-robin order.

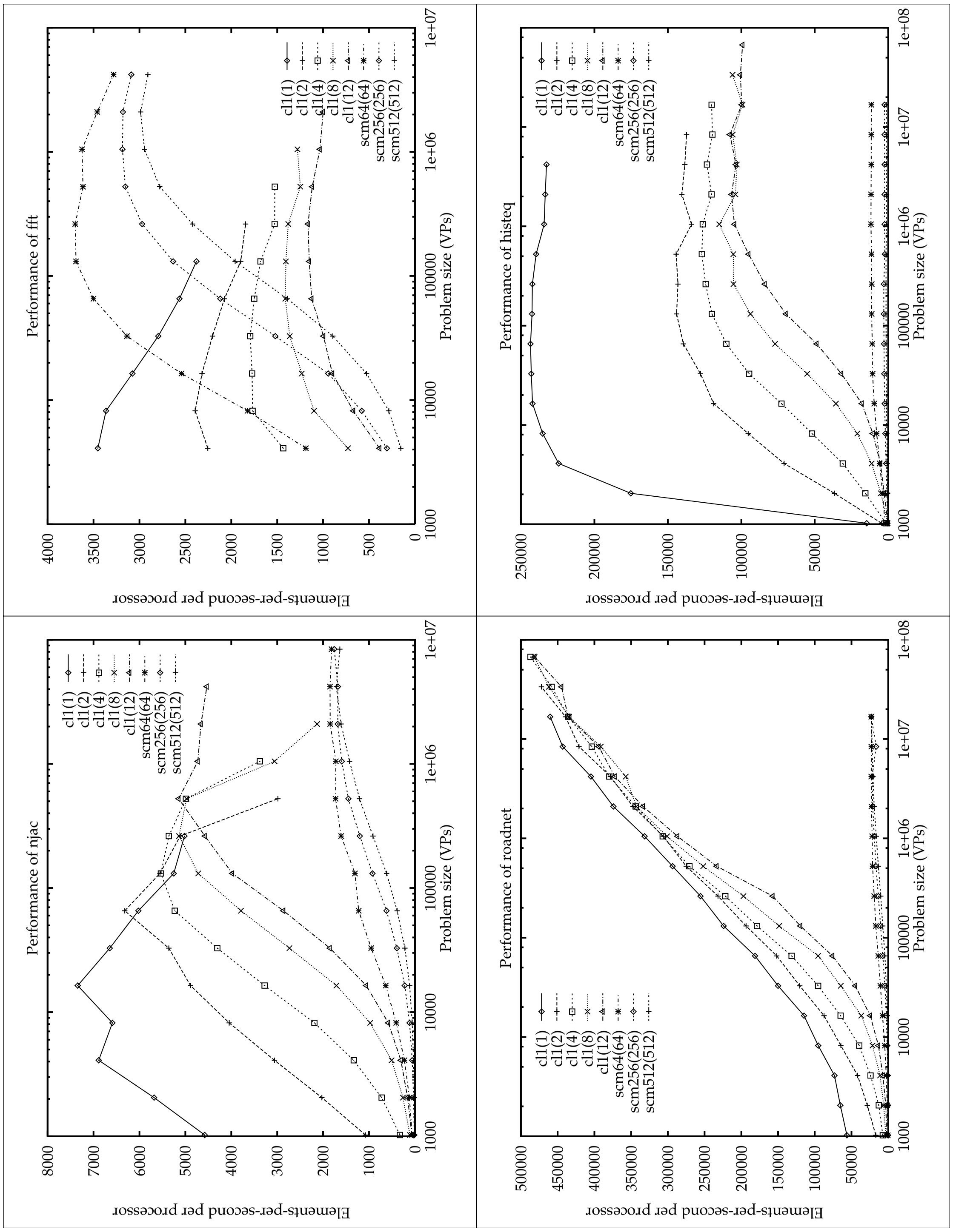


Figure 7.13a: Cluster and CM5 Performance: Elements Per Second Per Processor (Part 1)

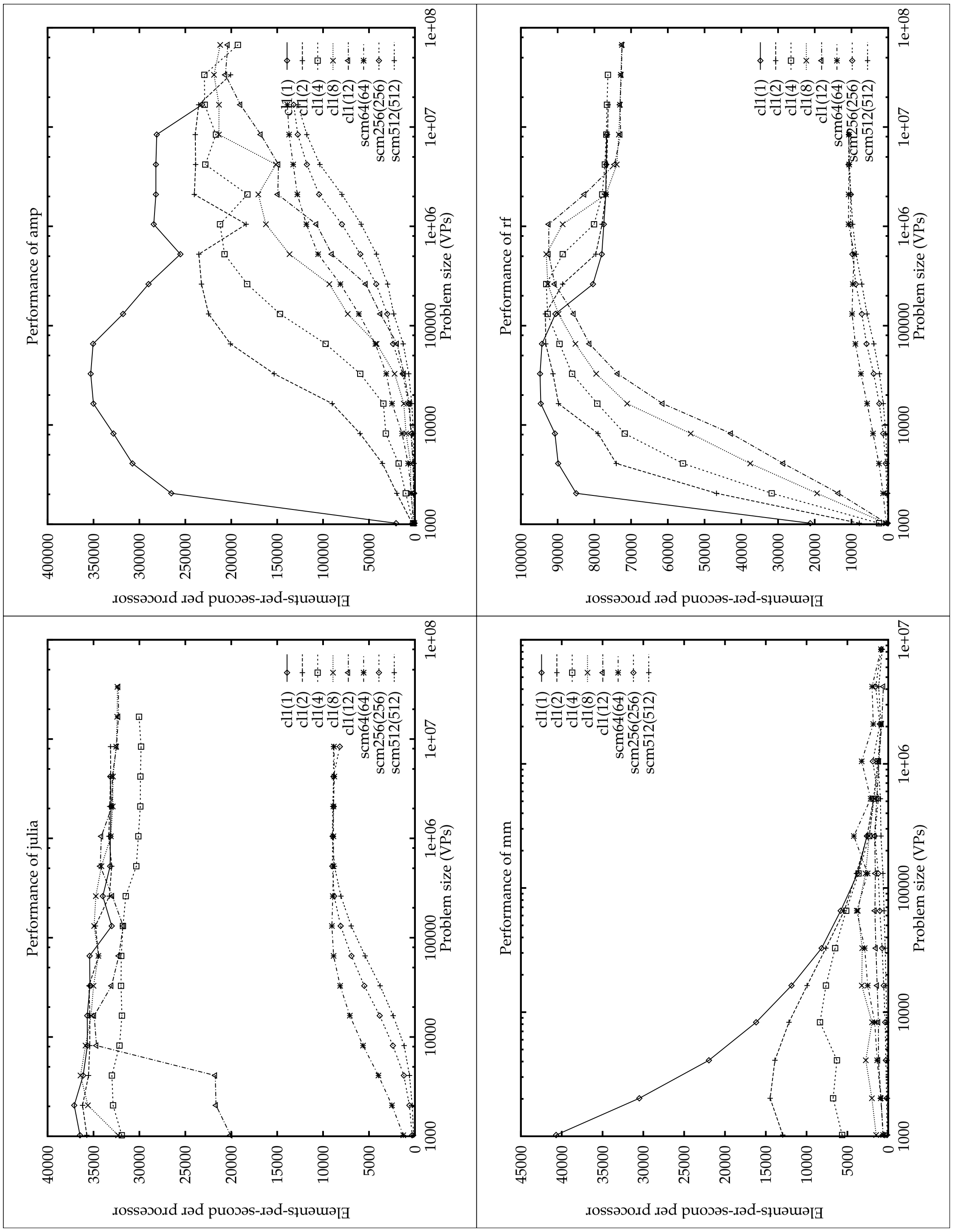


Figure 7.13b: Cluster and CM5 Performance: Elements Per Second Per Processor (Part 2)

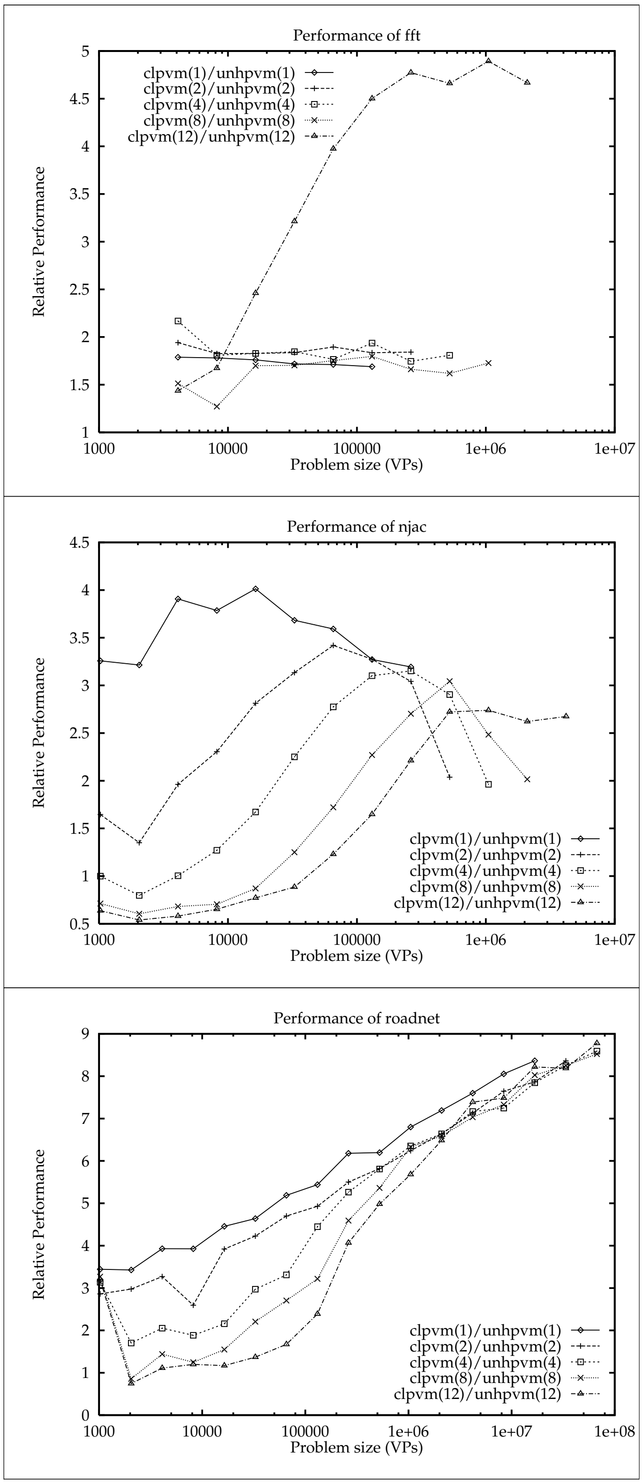


Figure 7.14: Cluster Relative Performance pC* / UNH C*

forms slightly faster than UNH when the reductions are removed; on the larger problems, pC*'s advantage returns closer to the factor of 3 improvement.

In the final benchmark, roadnet, the effect of the context optimizations of chapter 3 is clearly shown, with pC* rocketing above UNH C* at approximately eight times faster for the larger problem sizes.

CHAPTER 8
CONCLUSIONS

> Parallel computing is founded upon the premise that, if one worker can dig a post-hole in sixty seconds, sixty workers can dig a post-hole in one second.
> — Origin unknown

In this dissertation, we have considered many issues in the implementation of a runtime system for data parallel languages on stock networked workstations. We have supported our observations with extensive experimentation throughout the text, both in small programs designed to carefully test particular issues such as global-to-local address conversions and data access patterns, and in more complex programs which verify that the material described herein integrates well in a complete system.

Among the contributions of this dissertation we include:

- A heightened awareness of performance implications of local/global address conversion, and the way that distribution decisions can affect this (chapter 3). The techniques developed also allow us to use uniformly a cache-sensitive access pattern throughout the entire runtime system.

- A novel method of encoding the lists of active processors in C* programs, through run-length encoding, which saves space (up to 99% of a straightforward charmap encoding) and time by not requiring checks on each inactive position in a shape (chapter 3).

- A framework for portable but efficient communications support for the C* language (chapter 4). The approach described here should also work for other parallel languages which require extensive runtime support.

- A heuristic to measure the success rate of a runtime test and avoid performing the test when the test itself overwhelms the benefits that its result can enable (chapter 5). We apply the test to detect colliding general communications, yielding in some common cases a four-fold performance improvement.

- A method of handling arbitrary grid communications over block distributed shapes or arrays, which has extremely low overhead and is competitive with optimized special-case implementations of communications on shapes with one or two dimensions (chapter 6).

We have found that careful and considered design of one part of the system often yields opportunities for additional optimizations in a related part. For example, finding a method of uniformly walking all data in a strict sequential, contiguous order—desired independently to elicit good cache behavior—permits us to use an improved encoding of contexts, saving both space and time. Similarly, the communication handler functions—required for system correctness given limitations on network buffering—will permit an overlap of communication with computation by allowing the system to proceed with future computations during the latency periods of a communication whose result is not needed immediately.

We have proved the performance of the pC* system by contrasting it with both optimized sequential C solutions of a set of eight problems, and the latest C* implementation on a specially-designed supercomputer, the CM5. Except for algorithms which admit highly optimized sequential implementations not conducive to parallelization, a four-processor pC* system generally meets or exceeds the performance of the optimized sequential algorithm. A twelve workstation cluster connected with Ethernet outperforms a 64 node supercomputer with 5GB/sec interconnection on three of the eight benchmarks.

The portability of the system has been proved by giving performance results on networked Unix workstations, a symmetric multiprocessor, and a distributed memory multiprocessor. The system has been ported to a total of eight hardware platforms using five inter-process communications mechanisms. Porting the system is a straightforward task, taking roughly one programmer-day for each of the last three targets.

The methods described in this dissertation are applicable to a variety of parallel systems, not just C*. Many of the issues apply directly to the proposed Data Parallel C Extensions (Numerical C Extensions Group of X3J11, 1994). Others, especially the communications optimizations, can be integrated into any data-parallel system, such as Fortran 90 (Adams *et al.*, 1992) or High Performance Fortran (HPF Forum, 1993). The resulting synergism will increase the performance of compilers that perform extensive compile-time analyses by decreasing the performance gap when the analyses are unsuccessful, due to lacunae in the source program.

APPENDIX A
CODE FOR GRID COMMUNICATION

A little inaccuracy sometimes saves tons of explanation.
— Saki (H.H. Munro), *The Square Egg* (1924) "Clovis on the Alleged Romance of Business"

This appendix contains verbatim source for the grid algorithm described in chapter 6. To reduce space and confusion, only the material pertinent to the grid send operation is included, and conditionally compiled code relating to benchmarking or debugging has been removed.

`Id: ngridrw.c,v 2.33 1995/12/15 00:21:01 pab Exp`

A.1 Data types and accessors

```
/* Dimensions in the n-dimensional loop we emulate during grid operations
 * can be of one of these types */
typedef enum GridShiftType {
   ST_noshift,         /* Group of axes that don't have shifts */
   ST_block,           /* Axis has shift, but is entirely local */
   ST_distributed      /* Axis has shift, with data across two rnodes */
} GridShiftType;

/* Essential information about the boundaries of the n-dimensional
 * shift walk loop.  Note that number of split points depends on the
 * distribution of the axis, and can't be predetermined; it's
 * dynamically allocated during setup. */
typedef struct GridShiftAxisInfo {
   int axis;           /* Axis this boundinfo applies to */
   int shift;          /* Amount of shift along axis */
   int idelta;         /* Change in offset when incrementing by 1 */
   int wdelta;         /* Change in offset when wrapping back to llim */
   int walked;         /* Have we walked this axis during skip? */
   int wrapsplit;      /* Index where torus wraps */
   int cnt;            /* Upper limit position in count */
   int nsplit;         /* Number of splits along axis */
   int idx;            /* Current loop index */
   int * split;        /* Split limit points */
} GridShiftAxisInfo;
#define GSIData(_gsi, _k) ((_gsi)->sinfo + (_k))
#define GSIaxis(_gsi, _k) (GSIData(_gsi, _k)->axis)
#define GSIshift(_gsi, _k) (GSIData(_gsi, _k)->shift)
```

```
#define GSIincrdelta(_gsi, _k) (GSIData(_gsi, _k)->idelta)
#define GSIwrapdelta(_gsi, _k) (GSIData(_gsi, _k)->wdelta)
#define GSIwalked(_gsi, _k) (GSIData(_gsi, _k)->walked)
#define GSIcnt(_gsi, _k) (GSIData(_gsi, _k)->cnt)
#define GSInsplit(_gsi, _k) (GSIData(_gsi, _k)->nsplit)
#define GSIsplit(_gsi, _k) (GSIData(_gsi, _k)->split)
#define GSIidx(_gsi, _k) (GSIData(_gsi, _k)->idx)
#define GSIwrapsplit(_gsi, _k) (GSIData(_gsi, _k)->wrapsplit)

typedef struct GridShiftInfo {
   int validp;         /* Nonzero iff info set is initialized */
   int irnode;         /* Initial rnode */
   int irshift;        /* Initial rshift */
   int inib;           /* Initial IB size */
   int inoob;          /* Initial OOB size */
   int nshift;         /* Number of active shift axes */
   int mulfact;        /* Number of positions in highest nonshift set */
   int hascomm;        /* If nonzero, grid shifts along dist. axis */
   int ninbounds;      /* Number of positions in bounds */
   PCS__Shape shp;     /* Shape the grid is for */
   GridShiftAxisInfo sinfo [PCS__MAX_RANK]; /* Info about shift axes */
} GridShiftInfo;
#define GSIvalid(_gsi) ((_gsi)->validp)
#define GSIirnode(_gsi) ((_gsi)->irnode)
#define GSIirshift(_gsi) ((_gsi)->irshift)
#define GSIinoob(_gsi) ((_gsi)->inoob)
#define GSIinib(_gsi) ((_gsi)->inib)
#define GSInshift(_gsi) ((_gsi)->nshift)
#define GSImulfact(_gsi) ((_gsi)->mulfact)
#define GSIhascomm(_gsi) ((_gsi)->hascomm)
#define GSIninbounds(_gsi) ((_gsi)->ninbounds)
#define GSIshape(_gsi) ((_gsi)->shp)

/* For pre-determining how we're going to handle certain blocks of data */
typedef enum MoveMode {
   MM_ignore,          /* Ignore it */
   MM_blockzero,       /* Fill block with 0s */
   MM_blockmove,       /* Move over in one chunk */
   MM_elementzero,     /* Fill with 0s, element-by-element */
   MM_elementdoop      /* Perform doop element-by-element */
} MoveMode;
```

A.2 Loop Initialization

```
/* This function builds the loop emulator bounds and split points
 * associated with a particular grid shift on a particular shape. */
static int
```

```
setup_grid_bounds (PCS__Shape shp, /* Shape we're operating on */
                   int * offset, /* Offsets for shift */
                   int sign,    /* Direction of shift */
                   GridShiftInfo * sip) /* Where boundary info goes */
{
   int ashift;          /* Axis shift, from offset * sign */
   int nshift;          /* Number of dimensions in shift loop */
   int mulfact;         /* Scaling factor, combining nonshift axes */
   int noob;            /* Number of OOB elements starting region */
   int ninbounds;       /* Number of inbounds elements over all dims */
   int k;               /* Index over axes */
   int rnode;           /* Remote node we start with */
   int rshift;          /* Shift corresponding to rnode */
   int bcol;            /* Which block along distributed axis are we at */
   int blimit;          /* Number elements along axis on current node */
   PCS__shape_pernode * rpn; /* Information about dist. on remote node */
   int axis;            /* Axis along which shift occurs */
   int llim;            /* Lower limit of comm */
   int ulim;            /* Upper limit of comm */

   nshift = 0;
   mulfact = 1;
   noob = 0;
   ninbounds = 1;

   /* Mark the info valid.  Whoever passed this in should have cleared
    * the valid field at the start.  The field is used to detect
    * whether there is dynamic memory in the structure (for split
    * points) which needs to be freed later. */
   assert (! GSIvalid (sip));
   GSIvalid (sip) = 1;

   /* Start with our node; if nothing shifts, we keep this, otherwise
    * we adjust it for the axes along which we shift.  The result is
    * the node that owns the first position we're looking at. */
   rnode = PCS__nodenum;
   GSIhascomm (sip) = 0;

   /* Loop through all the axes of the shape, classifing each one as a
    * shift or nonshift.  Aggregate the adjacent nonshift ones
    * together, since we can treat them as a contiguous block.  For
    * shift ones, determine the lower and upper bounds for that
    * portion of the shift loop, as well as information required to
    * get the target processors if we cross out of the initial
    * node. */
   for (k = 0; k < PCS__ShpRank (shp); k++) {
      ashift = sign * offset [k];
      if (0 == ashift) {
         /* Accumulate all adjacent non-shift axes */
```

```
      mulfact *= PCS__ShpDimLocal (shp, k);
      continue;
   }
   if (abs (ashift) >= PCS__ShpDim (shp, k)) {
      /* Shifts beyond the extent are equivalent to shifts to
       * exactly the extent; doesn't matter which direction in this
       * case, all values will show up as out-of-bounds. */
      ashift = PCS__ShpDim (shp, k);
   }
   if (1 < mulfact) {
      /* Save a set of non-shift axes.  We associate these with the
       * lowest axis in the set, and set the range to cover the
       * whole thing with no skips.  We offset the range by the
       * DimAbove for the highest axis, so that the rshift change
       * is accurate. */
      assert (0 < k);
      GSIaxis (sip, nshift) = k-1;
      GSIshift (sip, nshift) = 0;
      GSInsplit (sip, nshift) = 1;
      /* Allocate 2 split points. */
      GSIsplit (sip, nshift) = malloc ((1+GSInsplit(sip, nshift))
                                       * sizeof (int));
      assert (NULL != GSIsplit (sip, nshift));
      GSIsplit (sip, nshift) [0] = PCS__ShpDimAbove (shp, k-1);
      GSIsplit (sip, nshift) [1] = GSIsplit (sip, nshift)[0] + mulfact;
      GSIincrdelta (sip, nshift) = PCS__ShpNumPerAxis (shp, k-1);
      GSIwrapdelta (sip, nshift) = 0;
      GSIidx (sip, nshift) = GSIsplit (sip, nshift) [0];
      GSIcnt (sip, nshift) = 1;
      ninbounds *= mulfact;
      mulfact = 1;
      nshift++;
   }
   GSIaxis (sip, nshift) = k;
   GSIshift (sip, nshift) = ashift;

   /* In the worst case distribution, we may cross an internal node
    * boundary DistNumBlocks-1 times.  This plus the two external
    * node boundaries determines the number of split points
    * needed. */
   GSIsplit (sip, nshift) = malloc ((1+PCS__ShpDistNumBlocks (shp, k))
                                    * sizeof (int));
   assert (NULL != GSIsplit (sip, nshift));

   /* Compute the lower and upper limits of the targets to which
    * our subgrid extent along axis k will map.  Truncate to the
    * ends of the full shape.  We'll use global index coordinates
    * rather than local ones, to aid in offset calculations. */
   llim = PCS__ShpDimAbove (shp, k) + ashift;
```

```
ulim = llim + PCS__ShpDimLocal (shp, k);
if (0 > llim) {
   llim = 0;
} else if (PCS__ShpDim (shp, k) < llim) {
   llim = PCS__ShpDim (shp, k);
}
if (0 > ulim) {
   ulim = 0;
} else if (PCS__ShpDim (shp, k) < ulim) {
   ulim = PCS__ShpDim (shp, k);
}
assert (llim <= ulim);
assert (0 <= llim);
assert (0 <= (llim - ashift));
assert (PCS__ShpDim (shp, k) >= ulim);
assert (PCS__ShpDim (shp, k) >= (ulim - ashift));

/* Set up the increments based on the axis and the out-of-bound
 * portion of the local axis */
GSIincrdelta (sip, nshift) = PCS__ShpNumPerAxis (shp, k);
GSIwrapdelta (sip, nshift) = (PCS__ShpDimLocal (shp, k)
                              - (ulim - llim))
   * GSIincrdelta (sip, nshift);

/* Compute which block along the axis the first source position
 * goes to---that's where we start.  First, subtract out any
 * component of this axis that is already reflected in rnode
 * because of our node number.  Then add the absolute
 * contribution from the walk. */
if (1 < PCS__ShpDistNumBlocks (shp, k)) {
   /* Retrench to basis for processors handling this axis. */
   bcol = (rnode / PCS__ShpDistPProd (shp, k))
      % PCS__ShpDistNumBlocks (shp, k);
   rnode -= bcol * PCS__ShpDistPProd (shp, k);
}
if (llim == ulim) {
   /* Nothing live on this node: just set up the walk range. */
   GSIsplit (sip, nshift) [0] = llim - ashift;
   GSIsplit (sip, nshift) [1] = ulim - ashift;
   GSInsplit (sip, nshift) = 1;
} else {
   /* Walk up to find the node that owns llim. */
   bcol = 0;
   blimit = PCS__ShpDistBlockSizes (shp, k) [0];
   while (llim >= blimit) {
      bcol++;
      assert (bcol < PCS__ShpDistNumBlocks (shp, k));
      blimit += PCS__ShpDistBlockSizes (shp, k) [bcol];
   }
```

```
      /* Adjust node number by the shift to get llim */
      rnode += bcol * PCS__ShpDistPProd (shp, k);

      /* Figure out all the break points along the axis: these are
       * computed by noticing when a target crosses a distribution
       * split, but are stored in terms of the source offset to
       * ease address translation */
      GSIsplit (sip, nshift)[0] = llim - ashift;
      GSInsplit (sip, nshift) = 1;

      /* As long as the upper limit exceeds what the range we've
       * looked at covers, we're going to cross into another
       * node. */
      while (ulim > blimit) {
         assert (GSInsplit (sip, nshift) <
                 (1+PCS__ShpDistNumBlocks (shp, k)));
         GSIsplit (sip, nshift) [GSInsplit (sip, nshift)] =
            blimit - ashift;
         ++GSInsplit (sip, nshift);
         ++bcol;
         assert (bcol < PCS__ShpDistNumBlocks (shp, k));
         blimit += PCS__ShpDistBlockSizes (shp, k) [bcol];
      }

      /* We end up in the final block; save the upper bound of the
       * transfer. */
      assert (GSInsplit (sip, nshift) <
              (1+PCS__ShpDistNumBlocks (shp, k)));
      GSIsplit (sip, nshift) [GSInsplit (sip, nshift)] =
         ulim - ashift;
   }

   /* Add the contribution of this axis to the in-bound region. */
   ninbounds *= (ulim - llim);

   /* If the shift is backwards, any OOB block from this axis
    * occurs at the start. */
   if (0 > ashift) {
      noob += GSIwrapdelta (sip, nshift);
   }

   /* Set up to start at the beginning of the in-bound region on
    * this axis */
   GSIidx (sip, nshift) = GSIsplit (sip, nshift) [0];
   GSIcnt (sip, nshift) = 1;

   /* There's communication if there's shifting along a distributed
    * block. */
```

```
   GSIhascomm (sip) |= PCS__ShpAxisIsDistributed (shp, k);

   nshift++;
}
/* Either the base mulfact is positive, or there's nothing on this
 * node. */
assert ((0 < mulfact) ||
        ((0 == mulfact) && (0 == PCS__ShpNumLocal (shp))));

/* Scale the values that are in blocks by the number of elements
 * per base block. */
ninbounds *= mulfact;

/* Compute the delta which we add to a local offset to get the
 * right offset for a remote node.  Note that the loop bounds and
 * index values are in global values along the axis.  Therefore,
 * the local offset is:
 * lo = sum (idx_k - Above_k{local}) * NPA_k{local}
 * and the remote offset will be:
 * ro = sum (idx_k + delta_k - Above_k{remote}) * NPA_k{remote}
 * We want to find rshift = ro - lo.  This value differs depending
 * on the remote node and on the axis.  Though we could precompute
 * some of the terms in (ro-lo), it'd be a pain to dynamically
 * allocate the array to hold them for each grid operation, so we
 * just recompute them at each stage.  (Note that if there is no
 * shift along an axis, both local and remote have the same NPA for
 * that axis, and so there's no component for that axis in rshift.)
 * */
rshift = 0;
rpn = PCS__ShpNodeLocalDist (shp, rnode);
for (k = 0; k < nshift; k++) {
   axis = GSIaxis (sip, k);
   rshift += (GSIidx (sip, k) + GSIshift (sip, k)
              - PCS__SPNAbove (rpn, axis)) * PCS__SPNNPA (rpn, axis)
      - (GSIidx (sip, k) - PCS__ShpDimAbove (shp, axis))
        * PCS__ShpNumPerAxis (shp, axis);
}

if (0 == ninbounds) {
   GSIinoob (sip) = PCS__ShpNumLocal (shp);
} else {
   GSIinoob (sip) = noob;
}
if (0 == ninbounds) {
   GSIinib (sip) = 0;
} else if (0 == nshift) {
   GSIinib (sip) = mulfact;
} else {
   GSIinib (sip) = (GSIsplit (sip, nshift-1)[GSIcnt (sip, nshift-1)]
```

```
                      - GSIsplit (sip, nshift-1)[0]) * mulfact;
   }
   GSImulfact (sip) = mulfact;
   GSInshift (sip) = nshift;
   GSIirnode (sip) = rnode;
   GSIirshift (sip) = rshift;
   GSIninbounds (sip) = ninbounds;
   GSIshape (sip) = shp;

   return ninbounds;
}
```

A.3 Region Search Support

Routines that find particular regions, given current position.

```
/* For restricted loop iterations, what type of sequence do we want to
 * stop at? */
typedef enum IBSkipTo {
   IBST_local,         /* Stop at blocks on this node */
   IBST_remote,        /* Any block not this node */
   IBST_remote_um_get, /* Any remote block not marked for get */
   IBST_remote_um_send /* Any remote block not marked for send */
} IBSkipTo;

/* Conditions that implement each of the above stopping cases. */
#define IBST_TestNode(_ibst,_rn) ( \
   (IBST_local == (_ibst)) ? (PCS__nodenum == (_rn)) : \
   (IBST_remote == (_ibst)) ? (PCS__nodenum != (_rn)) : \
   (IBST_remote_um_get == (_ibst)) ? ((PCS__nodenum != (_rn)) && \
                                   ! SGNgneed (_rn)) : \
   (IBST_remote_um_send == (_ibst)) ? ((PCS__nodenum != (_rn)) && \
                                    ! SGNsneed (_rn)) : \
   (assert (0), 0))

/* Jump to the next out-of-bounds region, updating offs to the proper
 * offset for that.  Returns the size of the oob region. */
PCS__INLINE static int
skip_to_grid_oob (GridShiftInfo * sip, /* Shift info */
                  int * offs)   /* Current offset */
{
   int k;
   int noob;

   k = GSInshift (sip) - 1;
   *offs += (GSIsplit (sip, k) [GSInsplit(sip, k)] - GSIidx (sip, k))
      * GSIincrdelta (sip, k);
   GSIidx (sip, k) = GSIsplit (sip, k) [0];
```

```
   GSIcnt (sip, k) = 1;
   noob = GSIwrapdelta (sip, k);
   while ((0 <= --k) &&
          (++GSIidx (sip, k) == GSIsplit (sip, k)[GSInsplit (sip, k)])) {
      noob += GSIwrapdelta (sip, k);
      GSIidx (sip, k) = GSIsplit (sip, k) [0];
      GSIcnt (sip, k) = 1;
   }
   return noob;
}

PCS__INLINE static int
skip_to_grid_inbound (
   IBSkipTo skipto, /* Criterion for accepting target */
   GridShiftInfo * sip, /* Information about shifts */
   int * offs, /* Current offset in local shape */
   int * rnodep, /* Node *offs maps to */
   int * rshiftp) /* Delta to get to target on rnode */
{
   int ib;             /* Size of in-bound block */
   int k;              /* Index to shift group being walked */
   int mink;           /* Minimum k visited, for setting walked flags */
   int clearforskip;   /* Is it OK to skip to split when we loop */
   int validr;         /* Is the rnode value correct */

   /* Note to all readers: This is probably the most terse and complex
    * function (along with skip_to_torus) in the entire pC* system.
    * There is not a single aspect of control flow that isn't critical
    * to correct behavior.  Modify at your own peril. */
   clearforskip = 1;
   validr = 1;
   k = mink = GSInshift (sip) - 1;
   do {
      /* See if we go to the next split point along the current axis */
      if (clearforskip ||
          (++GSIidx (sip, k) == GSIsplit (sip, k)[GSIcnt (sip, k)])) {
         if (GSIcnt (sip, k) < GSInsplit (sip, k)) {
            /* Internal split point.  Jump offset to split point, and
             * up the remote node as well. */
            *offs += (GSIsplit (sip, k)[GSIcnt (sip, k)]
                      - GSIidx (sip, k)) * GSIincrdelta (sip, k);
            GSIidx (sip, k) = GSIsplit (sip, k) [GSIcnt (sip, k)];
            GSIcnt (sip, k)++;
            assert (0 != GSIshift (sip, k));
            *rnodep += PCS__ShpDistPProd (GSIshape(sip),
                                          GSIaxis (sip, k));
            /* Normally, the resulting node is valid.  However, if
             * there are nodes with 0 elements along this axis between
             * nodes that have data, there will be adjacent split
```

```
            * points with the same value.  Those must be skipped
            * over, each one inducing another step of node number
            * along the axis.  We ensure the step is taken by
            * blocking the validity of the rnode, and setting
            * clearforskip true so we don't increment the index
            * during the loop back.  (We want clearforskip true
            * anyway, so at worst this means we move it out of the
            * conditional since it's always true when k==nshift-1.)
            * */
           validr = (GSIsplit (sip, k) [GSIcnt (sip, k)-1] <
                     GSIsplit (sip, k) [GSIcnt (sip, k)]);
           clearforskip = 1;
           if (validr && (k < (GSInshift (sip) - 1))) {
              /* Clear walked field, indicating we haven't checked
               * higher axes for this split region.  Restart at
               * highest axis. */
              while (k < (GSInshift (sip) - 1)) {
                 GSIwalked (sip, k) = 0;
                 k++;
              }
           }
        } else {
           /* Wrap at end of axis */
           *offs += (GSIsplit (sip, k)[GSIcnt (sip, k)]
                     - GSIidx (sip, k)) * GSIincrdelta (sip, k)
              + GSIwrapdelta (sip, k);
           GSIidx (sip, k) = GSIsplit (sip, k) [0];
           GSIcnt (sip, k) = 1;
           if (0 != GSIshift (sip, k)) {
              *rnodep -= (GSInsplit (sip, k) - 1) *
                 PCS__ShpDistPProd (GSIshape (sip), GSIaxis (sip, k));
           }
           /* Decrement down to the next lower axis so we adjust its
            * idx value.  Mark clearforskip false so we don't
            * short-circuit around the increment.  Mark validr false
            * to guarantee we re-enter the loop to execute the
            * increment. */
           clearforskip = 0;
           validr = 0;
           k--;
           if ((k < mink) && (0 <= k)) {
              /* First time at this axis: clear the walked field */
              GSIwalked (sip, k) = 0;
              mink = k;
           }
        }
     } else {
        assert (k < (GSInshift (sip) - 1));
        /* The increment of idx for this axis was done in the second
```

```
          * branch of the logical OR in the conditional we failed to
          * get here. */
         validr = 1;
         clearforskip = GSIwalked (sip, k);
         GSIwalked (sip, k) = 1;
         if (! clearforskip) {
            /* Gotta go back and look at things again.  Up axis to
             * highest, clearing the walked fields.  */
            while (++k < (GSInshift (sip) - 1)) {
               GSIwalked (sip, k) = 0;
            }
            clearforskip = 1;
         }
      }
   } while ((0 <= k) && ((! validr) ||
                         (! IBST_TestNode (skipto, *rnodep))));
   if (0 <= k) {
      PCS__shape_pernode * rpn; /* Distribution info for remote node */
      PCS__Shape shp;           /* Current shape */
      int axis;                 /* Axis for each loop */

      /* Compute the appropriate shift value added to local offsets to
       * get the remote offset for a particular node. */
      assert (validr);
      *rshiftp = 0;
      assert (0 <= *rnodep);
      assert (*rnodep < PCS__mesh_size);
      shp = GSIshape (sip);
      rpn = PCS__ShpNodeLocalDist (shp, *rnodep);
      assert (0 < PCS__SPNNumPos (rpn));
      for (k = 0; k < GSInshift (sip); k++) {
         axis = GSIaxis (sip, k);
         assert (0 <= axis);
         assert (axis < PCS__ShpRank (shp));
         /* See setup_grid_bounds for explanation of this formula */
         *rshiftp += (GSIidx (sip, k) + GSIshift (sip, k) -
                      PCS__SPNAbove (rpn, axis)) * PCS__SPNNPA (rpn, axis)
            - (GSIidx (sip, k) - PCS__ShpDimAbove (shp, axis))
              * PCS__ShpNumPerAxis (shp, axis);
      }
      k = GSInshift (sip) - 1;
      ib = (GSIsplit (sip, k)[GSIcnt (sip, k)] - GSIidx (sip, k))
         * GSIincrdelta (sip, k);
   } else {
      ib = 0;
   }
   return ib;
}
```

A.4 Grid Send

```
/* Do a grid write from srcp into destp, performing dest op= src, with
 * a shift given by offset.  Where the corresponding source position
 * is out of range and fillp is not null, use fillp. */
/* !!Begin PCS__defs!! */
void
PCS__grid_send (PCS__PvarPtr destp, /* Dest. pvar */
                PCS__PvarPtr srcp, /* Source pvar */
                PCS__PvarPtr fillp, /* Fill source for oob source */
                PCS__Type dtype, /* Type of dest */
                PCS__Type stype, /* Type of src */
                PCS__size_t size, /* Size of operand */
                PCS__RedOp op,  /* Reduction operator */
                int offset[])   /* Offsets */
/* !!End PCS__defs!! */
{
   int offs;           /* Local offset */
   PCS__ctx_rletype * ctxp; /* Pointer to context */
   int vplimit;        /* Maximum local vp index */
   int ctxvpi;         /* Context VP index */
   int ctxcnt;         /* Encoded context sequence */
   int nib, noob;      /* Number in and out of bounds for block */
   PCS__Shape shp;     /* Shape being walked */
   GridShiftInfo nsi;  /* Negative shift summary info */
   GridShiftInfo psi;  /* Positive shift summary info */
   int rnode, rshift;  /* Remote node and offset delta */
   MoveMode fillmode;  /* How to move from *fillp to *srcp */
   MoveMode sendmode;  /* How to move from *srcp to message */
   MoveMode recvmode;  /* How to move from message to *destp */
   PCS__DoopFunction * movedoop; /* Function to perform moves */
   PCS__DoopFunction * opdoop; /* Cached function to perform doop */

   ENTER_FUNCTION;

   /* Invalidate skip information, so we don't free unallocated
    * pointers */
   GSIvalid (&nsi) = GSIvalid (&psi) = 0;
   shp = PCS__PPshape (srcp);
   vplimit = PCS__ShpNumLocal (shp);
   movedoop = PCS__lookup_doop (PCS__NOP, dtype, stype);
   opdoop = PCS__lookup_doop (op, dtype, stype);

   /* Under normal circumstances, the source and dest are disjoint, so
    * we don't buffer local stuff, but just stuff it right where it's
    * supposed to go.  Of course, if they aren't disjoint, that'll
    * break big-time, so we make sure they are. */
   if (PCS__PPdata (destp) == PCS__PPdata (srcp)) {
      PCSRTMemMark PCS__cplrtemp_mark; /* Temporary mempool marker */
```

```
      PCS__Pvar tvar;            /* Created temporary value */
      PCS__PvarPtr tvarp;        /* Pointer to tvar */
      int i;                    /* Index over local elements of dshp */

      /* Mark the current state of the temporary mem pool.  Allocate a
       * temporary which we can use for the source.  Copy the original
       * destination into the temporary.  Call ourselves with the same
       * arguments except the source.  Free the temporary, and
       * return. */
      PCS__cplrtemp_mark = PCS__RTMMark (PCS__RTMC_CompilerTemp);
      tvar = PCS__PvarAlloc (shp, size, PCS__RTMC_CompilerTemp);
      PCS__PPSetPointTo (tvarp, PCS__PVdata (tvar), tvar);
      if (PCS__PPstride (srcp) == PCS__PPstride (tvarp)) {
         assert (PCS__PPstride (tvarp) == size);
         memcpy (PCS__PPdata (tvarp), PCS__PPdata (srcp), vplimit * size);
      } else {
         for (i = PCS__ShpNumLocal (shp) - 1; i >= 0; i--) {
            movedoop (PCS__PPelement (tvarp, i), PCS__PPelement (srcp, i),
                      size);
         }
      }
      PCS__grid_send (destp, tvarp, fillp, dtype, stype, size, op,
                      offset);
      PCS__RTMReclaim (PCS__cplrtemp_mark, PCS__RTMC_CompilerTemp);
      LEAVE_FUNCTION;
      return;
   }

   if (PCS__PPshape (destp) != shp) {
      PCS__Fatal ("grid_send: Destination shape doesn't match"
                  "source shape.\n");
   }
   if ((! PCS__is_null_pvar_ptr (fillp)) &&
       (PCS__PPshape (fillp) != shp)) {
      PCS__Fatal ("grid_send: Fill shape doesn't match source shape.\n");
   }


   /* Set up the operation modes for fills and local receives, so we
    * don't have to check these in the body of the loop. */
   if (PCS__is_null_pvar_ptr (fillp)) {
      fillmode = MM_ignore;
   } else {
      if ((PCS__PPstride (fillp) == PCS__PPstride (destp)) &&
          (PCS__PPstride (fillp) == size)) {
         fillmode = MM_blockmove;
      } else {
         fillmode = MM_elementdoop;
      }
```

```
}
/* Mode for applying data from source or incoming message to
 * destination.  Must be conservative (message will [probably]
 * always be contiguous, but srcp might not).  If source and dest
 * types aren't the same, they might be different sizes, so we need
 * to ensure conversions are done. */
if ((PCS__NOP == op) && (dtype == stype) &&
    (PCS__PPstride (destp) == PCS__PPstride (srcp)) &&
    (PCS__PPstride (destp) == size)) {
   recvmode = MM_blockmove;
} else {
   recvmode = MM_elementdoop;
}

/* Mode for copying from source area into outging message buffer */
if (PCS__PPstride (srcp) == size) {
   sendmode = MM_blockmove;
} else {
   sendmode = MM_elementdoop;
}

CSafety_OOB_Reset ();

setup_grid_bounds (shp, offset, 1, &psi);

if (0 == GSInshift (&psi)) {
   opassign_var (destp, srcp, recvmode, dtype, stype, size, op);
   free_gridshiftinfo (&psi);
   LEAVE_FUNCTION;
   return;
}

if (GSIhascomm (&psi)) {
   init_addsdat (0, destp, dtype, stype, recvmode, op, size);
   assert (ASD_datactx != asdtype); /* This is only for gets */

   /* Walk through sending off data to the other side. */
   offs = GSIinoob (&psi);
   nib = GSIinib (&psi);
   rnode = GSIirnode (&psi);
   rshift = GSIirshift (&psi);
   ctxvpi = 0;
   ctxp = PCS__ShpContext (shp);
   PCS__ctx_nextseq (ctxcnt, ctxp, ctxvpi, vplimit);
   while (offs < vplimit) {
      if (PCS__nodenum != rnode) {
         int toffs = offs;   /* Mutable offset value */

         assert (0 <= rnode);
```

```
            assert (rnode < PCS__mesh_size);
            SGNsused (rnode) = 1;
            while (0 < nib) {
               int cnt;
               PCS__ctx_skiptovpi (&ctxcnt, &ctxp, &ctxvpi, vplimit,
                                   toffs);
               if (0 > ctxcnt) {
                  /* Active sequence: package up to min (bsize, ctxcnt) */
                  cnt = -ctxcnt;
                  if (cnt > nib) {
                     cnt = nib;
                  }
                  addsdata (srcp, toffs, cnt, sendmode, movedoop,
                            rnode, toffs + rshift, size);
               } else {
                  /* Inactive sequence: skip to min (nib, ctxcnt) */
                  cnt = ctxcnt;
                  if (cnt > nib) {
                     cnt = nib;
                  }
               }
               nib -= cnt;
               toffs += cnt;
            }
         }
         nib = skip_to_grid_inbound (IBST_remote, &psi, &offs, &rnode,
                                     &rshift);
      }
      reset_shift_info (&psi);

      /* Send the final packets on their way */
      flush_addsdat ();

      /* Now walk looking the other way, and see who's going to be
       * sending us something, so we're sure we've finished */
      setup_grid_bounds (shp, offset, -1, &nsi);
      offs = GSIinoob (&nsi);
      nib = GSIinib (&nsi);
      rnode = GSIirnode (&nsi);
      while (offs < vplimit) {
         if ((PCS__nodenum != rnode) && ! SGNsneed (rnode)) {
            SGNsneed (rnode) = 1;
            sdat_nleft++;
         }
         nib = skip_to_grid_inbound (IBST_remote_um_send, &nsi, &offs,
                                     &rnode, &rshift);
      }
      sdat_flready = 1;
   }
}
```

```
/* Walk handling the local stuff */
offs = GSIinoob (&psi);
nib = GSIinib (&psi);
rnode = GSIirnode (&psi);
rshift = GSIirshift (&psi);
ctxvpi = 0;
ctxp = PCS__ShpContext (shp);
PCS__ctx_nextseq (ctxcnt, ctxp, ctxvpi, vplimit);
while (offs < vplimit) {
   if (PCS__nodenum == rnode) {
      int toffs = offs;      /* Mutable offset value */

      /* Make sure everybody will be in range */
      if (toffs + nib >= vplimit) {
         nib = vplimit - toffs;
      }
      assert (0 <= toffs);
      assert (toffs < vplimit);
      assert (0 <= toffs + rshift);
      assert (toffs + rshift < vplimit);

      /* Send is contexted from sourcep position */
      while (0 < nib) {
         int cnt;

         PCS__ctx_skiptovpi (&ctxcnt, &ctxp, &ctxvpi, vplimit, toffs);
         if (0 > ctxcnt) {
            /* Active sequence: package up to min (nib, ctxcnt) */
            cnt = -ctxcnt;
            if (cnt > nib) {
               cnt = nib;
            }
            switch (recvmode) {
               case MM_blockmove:
                  memcpy (PCS__PPelement (destp, toffs + rshift),
                          PCS__PPelement (srcp, toffs), size * cnt);
                  break;
               case MM_elementdoop: {
                  int i;
                  char * dp, * sp;
                  int ddp, dsp;

                  dp = PCS__PPelement (destp, toffs + rshift);
                  sp = PCS__PPelement (srcp, toffs);
                  ddp = PCS__PPstride (destp);
                  dsp = PCS__PPstride (srcp);
                  i = cnt;
                  while (0 < i--) {
```

```
                        opdoop (dp, sp, size);
                        dp += ddp;
                        sp += dsp;
                     }
                     break;
                  }
                  default:
                     assert (0);
               }
            } else {
               /* Inactive sequence: skip to min (nib, ctxcnt) */
               cnt = ctxcnt;
               if (cnt > nib) {
                  cnt = nib;
               }
            }
            nib -= cnt;
            toffs += cnt;
         }
      }
      nib = skip_to_grid_inbound (IBST_local, &psi, &offs, &rnode,
                                  &rshift);
   }

   /* Warn about any attempts to send out-of-bounds. */
   if (! CSafety_OOB_Ignore ()) {
      offs = 0;
      ctxp = PCS__ShpContext (shp);
      ctxvpi = 0;
      PCS__ctx_nextseq (ctxcnt, ctxp, ctxvpi, vplimit);
      reset_shift_info (&psi);
      noob = GSIinoob (&psi);
      while (offs < vplimit) {
         assert (0 <= noob);
         if (0 < noob) {
            /* We have a sequence of OOB positions who we "send" to.
             * If any are active, this is illegal, so do a warning. */
            if (offs + noob > vplimit) {
               noob = vplimit - offs;
            }
            while (0 < noob) {
               int cnt;

               PCS__ctx_skiptovpi (&ctxcnt, &ctxp, &ctxvpi, vplimit,
                                   offs);
               if (0 > ctxcnt) {
                  /* Active sequence.  Skip to min (ctxcnt, noob) and
                   * warn. */
                  cnt = -ctxcnt;
```

```
                if (cnt > noob) {
                   cnt = noob;
                }
                PCS__OOBWarn ("grid send", PCS__current, NULL);
             } else {
                /* Inactive sequence.  Skip to min (ctxcnt, noob). */
                cnt = ctxcnt;
                if (cnt > noob) {
                   cnt = noob;
                }
             }
             noob -= cnt;
             offs += cnt;
          }
       }
        noob = skip_to_grid_oob (&psi, &offs);
    }
 }

 /* Handle any OOB reception actions. */
 if (MM_ignore != fillmode) {
    if (GSIvalid (&nsi)) {
       /* Already created nsi during send walk */
       reset_shift_info (&nsi);
    } else {
       /* Create nsi so we can see what incoming sends come from OOB */
       setup_grid_bounds (shp, offset, -1, &nsi);
    }
    offs = 0;
    noob = GSIinoob (&nsi);
    while (offs < vplimit) {
       assert (0 <= noob);
       if (0 < noob) {
          /* We have a sequence of positions who are "sent" to by
           * OOB.  Do a context insensitive move from the fill value
           * into the region. */
          if (offs + noob > vplimit) {
             noob = vplimit - offs;
          }
          switch (fillmode) {
             case MM_blockmove:
                /* Fills in send operations are uncontexted.  Blow
                 * the data in. */
                memcpy (PCS__PPelement (destp, offs),
                        PCS__PPelement (fillp, offs), size * noob);
                break;
             case MM_elementdoop: {
                int i;
                char * dp, * sp;
```

```
                  int ddp, dsp;

                  dp = PCS__PPelement (destp, offs);
                  sp = PCS__PPelement (fillp, offs);
                  ddp = PCS__PPstride (destp);
                  dsp = PCS__PPstride (fillp);
                  for (i = 0; i < noob; i++) {
                     opdoop (dp, sp, size);
                     dp += ddp;
                     sp += dsp;
                  }
                  break;
               }
               default:
                  assert (0);
            }
         }
         offs += noob;
         noob = skip_to_grid_oob (&nsi, &offs);
      }
   }

   /* If we're communicating, wait for the last of the incoming data */
   if (GSIhascomm (&psi)) {
      finish_sdat ();
   }

   /* Free any dynamic memory allocated during gsi setup, looking at
    * the valid flag to see if there was any. */
   free_gridshiftinfo (&nsi);
   free_gridshiftinfo (&psi);
   LEAVE_FUNCTION;
   return;
}
```

APPENDIX B
C* BENCHMARK CODE

Where's the beef?

— Cliff Freeman (advertizing slogan for Wendy's Hamburgers; words spoken by Clara Peller)

This appendix contains source for the C* benchmarks used in chapter 7.

B.1 Fast Fourier Transform

```
/* Id: fft.cs,v 1.1 1996/01/09 15:23:21 pab Exp
 *
 * FFT: Implements a 1 dimensional complex FFT.  The real and
 *      imaginary parts are presented separately in two poly floats.
 *      The length of the data array is n, and n is 2^p.  */

#include <stdio.h>
#include <stdlib.h>
#include <cscomm.h>
#include <math.h>
#include <assert.h>
#include <cm/timers.h>

#ifndef M_PI
#define M_PI            3.14159265358979323846
#endif /* M_PI */

shape []Shape1d;

void fft_1d (float:Shape1d *realpart,
             float:Shape1d *imagpart,
             bool inversep)
{
  int i;              /* General index value */
  int spacing;        /* Stride in butterfly loop */
  int iteration = 0;  /* Which iteration of loop? */
  int nbits;          /* Number of bits in loop mask */
  double pi;          /* Value of pi for forward/invert */
  float:current sin_factor; /* Trig factors of pi/spacing */
  float:current cos_factor;
  float:current real_assoc; /* Temps for communicated values */
  float:current imag_assoc;
```

```
float:current ftemp; /* Temp for source for trig factors */
unsigned int:current name; /* Bit mask for partners */
unsigned int:current name_shift;
unsigned int:current assoc;
bool:current assoc_flag; /* Which type of partner? */

/* Set nbits to floor(log_2 (dimof (current, 0))) */
i = dimof (current, 0);
nbits = 0;
while (1 < i) {
  ++nbits;
  i >>= 1;
}
if (dimof (current, 0) != (1 << nbits)) {
  fprintf (stderr, "fft1d: Error: Incoming shape must have power-of-2"
           "dimension (has %d)\n", dimof (current, 0));
  exit (1);
}

everywhere {

  /* Reverse the bits in the processor numbers */
  name_shift = name = pcoord(0);
  assoc = 0;
  for (i = 0; i < nbits; i++) {
    assoc = (assoc << 1) | (name_shift & 1);
    name_shift >>= 1;
  }
  name_shift = name;

  [assoc] *realpart = *realpart;
  [assoc] *imagpart = *imagpart;

  pi = inversep ? -M_PI : M_PI;

  /* top of butterfly loop */
  for (spacing = 1; spacing < dimof (current, 0); spacing = 2*spacing) {
    iteration++;

    /* assign associate processor */
    where (name_shift % 2) {
      assoc_flag = 1;
      assoc = name - spacing;
    } else {
      assoc_flag = 0;
      assoc = name + spacing;
    }

    /* exchange data between associated processors */
```

```
      [assoc] real_assoc = *realpart;
      [assoc] imag_assoc = *imagpart;

      /* prepare data in primary processors */
      where (assoc_flag) {
        ftemp = *realpart;
        *realpart = real_assoc;
        real_assoc = -ftemp;
        ftemp = *imagpart;
        *imagpart = imag_assoc;
        imag_assoc = -ftemp;
      }

      /* Obtain phase factors.  For FFT inversion, the value of pi has
       * been negated so the sign difference in the sin component for
       * the fft inverse formula is effected. */
      ftemp = (pi / spacing) * (name % spacing);
      cos_factor = cos (ftemp);
      sin_factor = sin (ftemp);

      *realpart += (cos_factor * real_assoc) + (sin_factor * imag_assoc);
      *imagpart += (cos_factor * imag_assoc) - (sin_factor * real_assoc);

      name_shift >>= 1;
    }

    /* Normalize for inverse transform */
    if (inversep) {
      float tmp = 1.0F / dimof(current, 0);
      *realpart *= tmp;
      *imagpart *= tmp;
    }
  }
}

main(int argc, char **argv)
{
  int len;
  int loglen;

  len = 65536;
  if (1 < argc) {
    len = atoi (argv [1]);
  }

  loglen = 0;
  while (1 < len) {
    ++loglen;
    len >>= 1;
```

```
  }
  len = (1 << loglen);

  allocate_shape(&Shape1d, 1, len);

  with (Shape1d) everywhere {
    float:current realpart, imagpart; /* Data we're operating on */
    float:current rp0, ip0; /* Original values, for comparisons */

    rp0 = (prand () % 10000) / 100.0 - 50.0;
    ip0 = (prand () % 10000) / 100.0 - 5.0;

    realpart = rp0;
    imagpart = ip0;
    CM_timer_clear(0);
    CM_timer_start(0);

    fft_1d(&realpart, &imagpart, 0);
    fft_1d(&realpart, &imagpart, 1);

    CM_timer_stop(0);

    printf ("# Maximum difference is (%g, %g)\n",
            >?= fabs (rp0 - realpart),
            >?= fabs (ip0 - imagpart));
    printf ("%10.4f %10.4f %10.4f # fft %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), len, positionsof (Shape1d),
            positionsof (physical));
  }
  deallocate_shape (&Shape1d);
}
```

B.2 Histogram Equalization

```
/* Id: histeq.cs,v 1.1 1996/01/09 15:23:21 pab Exp
 * Created: Tue Jun  6 13:36:36 1995
 * Peter A. Bigot (pab@clotho)
 * Last Revised:
 *
 * Description:
 * C* version of the histogram equalization filter benchmark.
 *
 * Update Information:
 * ----------------------------------------
 * End of updates
 */
```

```
/* ------------- */
/* Include Files */

#include <stdio.h>              /* Standard input/output routines */
#include <stdlib.h>             /* Standard library routines */
#include <assert.h>             /* Debugging assertion macro */
#include <cscomm.h>
#include <cm/timers.h>
#include <string.h>

/* ------------------------------- */
/* Constant and Macro Declarations */

/* ----------------- */
/* Type Declarations */

/* -------------------- */
/* Variable Definitions */

/* -------------------- */
/* Function Definitions */

/* Enhance an image by assigning intensities based on the frequency of
 * intensities in the original image.  Sets a new image. */
void
histogram_equalization (
   unsigned char:void * imp, /* Pointer to source image */
   unsigned char:shapeof(*imp) * newimage,
   unsigned int outpelrange)  /* Number of intensities in output image */
{
  shape [] PixelVal;
  unsigned char maxpel;

  with (shapeof (*imp)) everywhere {
    /* Find the maximum pixel value, and set up a shape to histogram
     * into. */
    maxpel = >?= *imp;
    allocate_shape (&PixelVal, 1, maxpel+1);
    with (PixelVal) everywhere {
      unsigned int:current hist;
      unsigned int:current histrc;
      unsigned char:current newpel;

      /* Count the number of times each pixel value appears in the
       * image. */
      hist = 0;
      with (shapeof (*imp)) everywhere {
        [*imp] hist += (int:current) 1;
      }
```

```
      /* Do a running sum of the histogram values, then normalize them
       * over the real pixel range, assigning each original pixel
       * value to a new pixel value in the normalized histogram based
       * on the midpoint of the histogram bins. */
      histrc = scan (hist, 0, CMC_combiner_add, CMC_upward, CMC_none,
                     CMC_no_field, CMC_exclusive);
      newpel = (unsigned char:current)
         (outpelrange * (histrc + hist/2.0)
          / (float:current) positionsof (shapeof (*imp)));
      /* Read the normalized pixel values from the "normalized"
       * histogram bins. */
      with (shapeof (*imp)) everywhere {
        *newimage = [*imp] newpel;
      }
    }
    deallocate_shape (&PixelVal);
  }
}

int main (int argc,    /* Number of command line arguments */
          char * argv []) /* Array of command line arguments */
{
  int nrows;           /* Rows in image */
  int ncols;           /* Columns in image */
  shape Image;         /* Image shape */

  nrows = ncols = 512;
  if (1 < argc) {
    nrows = atoi (argv [1]);
  }
  if (2 < argc) {
    ncols = atoi (argv [2]);
  }

  allocate_shape (&Image, 2, nrows, ncols);
  with (Image) everywhere {
    unsigned char:current img;
    unsigned char:current mfimg;

    img = 25 + (prand () %% 200);
    CM_timer_clear (0);
    CM_timer_start (0);
    histogram_equalization (&img, &mfimg, 256);
    CM_timer_stop (0);
    printf ("%12.5g %12.5g %12.5g # histeq %d %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), dimof (current, 0),
            dimof (current, 1), positionsof (current),
            positionsof (physical));
```

```
  }
  deallocate_shape (&Image);
  return (0);
}
```

B.3 Jacobi Iteration

```
/* Id: njac.cs,v 1.1 1996/01/09 15:23:21 pab Exp
 * Created: ??
 * Peter A. Bigot (pab@alecto)
 * Last Revised:
 *
 * Description:
 * Jacobi iteration benchmark.
 *
 * Update Information:
 * ----------------------------------------
 * End of updates
 */

/* ------------- */
/* Include Files */

#include <stdlib.h>
#include <stdio.h>
#include <cscomm.h>
#include <math.h>
#include <cm/timers.h>

shape [][] Field;

int
main (int argc, char * argv [])
{
  float delta;
  int niters;
  int maxiters;
  int nrow, ncol;

  ncol = nrow = 128;
  maxiters = 100;

  if (1 < argc) {
     nrow = atoi (argv [1]);
  }
  if (2 < argc) {
     ncol = atoi (argv [2]);
  }
```

```
  if (3 < argc) {
     maxiters = atoi (argv [3]);
  }

  allocate_shape (&Field, 2, nrow, ncol);
  with (Field) everywhere {
    float:Field field;

    where (0 == pcoord (1)) {
      field = 65.0F;
    } else where (dimof (current, 1)-1 == pcoord (1)) {
      field = 55.0F * pcoord (0) / dimof (current, 0);
    } else where (dimof (current, 0)-1 == pcoord (0)) {
      field = 55.0F;
    } else where (0 == pcoord (0)) {
      field = 0.0F;
    } else {
      field = 30.0F;
    }

    niters = maxiters;
    CM_timer_clear (0);
    CM_timer_start (0);
    while (0 < niters--) {
      float:current tmp;

      where ((0 < pcoord (0)) &&
             (0 < pcoord (1)) &&
             ((dimof (current, 0)-1) > pcoord (0)) &&
             ((dimof (current, 1)-1) > pcoord (1))) {
        tmp = ([.-1][.]field + [.+1][.]field +
               [.][.-1]field + [.][.+1]field) / 4.0F;
        /* Normally, we would use delta to determine convergence;
         * since this is a benchmark, we compute the necessary value,
         * but instead use iteration counts to determine how long we
         * should run. */
        delta = >?= fabs (field - tmp);
        field = tmp;
      }
    }
    CM_timer_stop (0);
    printf ("%12.5f %12.5f %12.5f # njac %d %d %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), nrow, ncol, maxiters,
            positionsof (current), positionsof (physical));
  }
  deallocate_shape (&Field);
}
```

B.4 Road Distance

```
/* Id: roadnet.cs,v 1.1 1996/01/09 15:23:21 pab Exp
 * Created: Wed Dec  6 08:17:52 1995
 * Peter A. Bigot (pab@alecto)
 * Last Revised:
 *
 * Description: Test to measure distance of points from roads: all
 * points within maxiters 4-connected steps from a "road" (marked as
 * 0s in the pvar) are labelled with the number of steps it took to
 * get there from some road point.
 *
 * Update Information:
 * ----------------------------------------
 * End of updates */

/* ------------- */
/* Include Files */

#include <stdio.h>     /* Standard input/output routines */
#include <stdlib.h>    /* Standard library routines */
#include <assert.h>    /* Debugging assertion macro */
#include <cm/timers.h> /* Timing support */

int main (int argc,    /* Number of command line arguments */
          char * argv []) /* Array of command line arguments */
{
  int height;          /* Rows in map */
  int width;           /* Columns in map */
  int maxiters;        /* How far away from road do we go */
  int i;               /* Index over iters */
  shape S;             /* Shape of map */

  height = width = 1024;
  maxiters = 10;
  if (1 < argc) {
    height = atoi (argv [1]);
  }
  if (2 < argc) {
    width = atoi (argv [2]);
  }
  if (3 < argc) {
    maxiters = atoi (argv [3]);
  }

  allocate_shape (&S, 2, height, width);
  with (S) everywhere {
    int:current map;
```

```
    /* Initialize the map: everywhere but a road is set to a maximum
     * value; road locations are set to 0.  For the benchmark, the
     * road network consists of an overlaid cross and X at the center
     * of the map, extending to all edges. */
    map = (int:current) (1+maxiters);
    where (((dimof (current, 0) / 2) == pcoord (0)) ||
           ((dimof (current, 1) / 2) == pcoord (1)) ||
           (pcoord (0) == pcoord (1)) ||
           ((dimof (current, 0) - 1 - pcoord (0)) == pcoord (1))) {
      map = 0;
    }

    CM_timer_clear(0);
    CM_timer_start(0);

    for (i = 0; i < maxiters; i++) {
      /* Restrict attention to regions known to be on the edge of
       * roads. */
      where (map == i) {
        int:current tmap;       /* Source map values */

        /* For each point known to be near a road, let its neighbors
         * know that they're at most one further away. */
        tmap = map + 1;
        where (0 < pcoord (0)) {
          [.-1][.] map <?= tmap;
        }
        where ((dimof (current, 0)-1) > pcoord (0)) {
          [.+1][.] map <?= tmap;
        }
        where (0 < pcoord (1)) {
          [.][.-1] map <?= tmap;
        }
        where ((dimof (current, 1)-1) > pcoord (1)) {
          [.][.+1] map <?= tmap;
        }
      }
    }

    CM_timer_stop(0);

    printf ("%10.4f %10.4f %10.4f # roadnet %d %d %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), height, width, maxiters,
            positionsof (current), positionsof (physical));

  }
  deallocate_shape (&S);
  return (0);
```

```
}
```

B.5 Amplitude Screener

```
/* Id: amp.cs,v 1.1 1996/01/09 20:40:52 pab Exp
 * amp.cs - scan-based amplitude screener
 */

#include <stdlib.h>    /* General support routines */
#include <cscomm.h>    /* Scan and communication routines */
#include <cm/timers.h> /* Timing support */

/* Maximum image value */
#define MAX_VAL 256

/* Detect all pixels which are more than threshold times the average
 * of their surrounding pixels within a wsize window in both
 * directions. */
int amp_screener (
   unsigned char:current *image, /* IN: Image data */
   int wsize,     /* IN: Window size */
   float threshold, /* IN: Threshold for brights */
   unsigned char min_threshold, /* IN: Min pixel threshhold for brights */
   bool:current *bright_return) /* OUT: Boolmask indicating brights */
{
  int:current iimage;  /* Temporary used for image window sums */
  int lwsize;          /* Extent of window strictly below center */
  int uwsize;          /* Extent of window strictly above center */

  /* Determine portion of window that falls to right of center pixel.
   * We allot (wsize/2) to left, and discount the center.  This
   * correctly handles even-sized windows (although there is a bias if
   * you use them). */
  lwsize = (wsize / 2);
  uwsize = wsize - lwsize - 1;
  everywhere {

    /* Determine the sum of the pixels in the wsizeXwsize window
     * centered on each pixel, using scan/subtract.  Note that we use
     * an integer-valued temporary to avoid problems with overflow. */
    iimage = scan ((int:current) *image, 1, CMC_combiner_add, CMC_upward,
                   CMC_none, CMC_no_field, CMC_inclusive);
    where (pcoord (1) >= wsize) {
      iimage -= [.][.-wsize] iimage;
    }
    iimage = scan (iimage, 0, CMC_combiner_add, CMC_upward,
                   CMC_none, CMC_no_field, CMC_inclusive);
    where (pcoord (0) >= wsize) {
```

```
      iimage -= [.-wsize][.] iimage;
    }

    /* iimage holds window sums, with sum appearing in lower right
     * (higher along axes) corner of the window.  Shift them back to
     * the center. */
    where ((pcoord (0) >= uwsize) &&
           (pcoord (1) >= uwsize)) {
      [.-uwsize][.-uwsize] iimage = iimage;
    }

    /* Subtract the center pixel from the total sum of the wsize,
     * yielding the sum of the surrounding pixels */
    iimage -= *image;

    /* Mark a bright if the window sum is valid (*center_pixel), the
     * image value meets the minimum threshold, and the image value is
     * more than threshold times the average of the surrounding
     * pixels. */

    /* Scale threshold to do average of surround when scaling center
     * value. */
    threshold /= wsize*wsize - 1;

    /* Only set brights within the active area; the edges are
     * non-bright. */
    where (((wsize / 2) <= pcoord (0)) &&
           ((wsize / 2) <= pcoord (1)) &&
           ((dimof (current, 0) - uwsize) > pcoord (0)) &&
           ((dimof (current, 1) - uwsize) > pcoord (1))) {
      *bright_return = (min_threshold < *image) &&
         ((threshold * iimage) < *image);
    } else {
      *bright_return = 0;
    }

    return += *bright_return;
  }
}

int main(int argc, char **argv)
{
  shape Imageshape;             /* Shape to use for images */
  int wsize;                    /* Window size for screening */
  int nreturns;                 /* Number of bright returns */
  float thresh;                 /* Threshold for brights */
  int num_cols;                 /* Columns in image */
  int num_rows;                 /* Rows in image */
```

```
num_cols = num_rows = 128;
wsize = 3;
thresh = 1.1;

/* Usage: amp nrows ncols winsize threshold */
if (1 < argc) {
  num_rows = atoi (argv [1]);
}
if (2 < argc) {
  num_cols = atoi (argv [2]);
}
if (3 < argc) {
  wsize = atoi (argv [3]);
}
if (4 < argc) {
  thresh = atof (argv [4]);
}

printf("# Amplitude Screener: Image [%d x %d],  ws: %d, thresh: %g\n",
       num_rows, num_cols, wsize, thresh);

Imageshape = allocate_shape(&Imageshape, 2, num_rows, num_cols);

with (Imageshape) {
  unsigned char:current image; /* Image being amp'd */
  bool:current bright;        /* Which positions are brights */

  everywhere {
    /* We could use randoms, but this varies with numbers of
     * processors, and would be difficult to match in the C version.
     * Use an image with intensity based on geometric value:
     * essentially, a mountain range laid out on grid interstices.
     * Not a good representation of reality, but the algorithm isn't
     * data-dependent anyway. */
    int:current gval;
    int vrange;

    gval = pcoord (0) + pcoord (1);
    vrange = MAX_VAL / 4;
    where (0 == (gval / vrange) %% 2) {
      image = vrange + (gval %% vrange);
    } else {
      image = vrange - (gval %% vrange);
    }
    gval = (dimof (current, 0) - pcoord (0)) + pcoord (1);
    where (0 == (gval / vrange) %% 2) {
      image += vrange + (gval %% vrange);
    } else {
      image += vrange - (gval %% vrange);
```

```
      }
    }


    CM_timer_clear(0);
    CM_timer_start(0);

    nreturns = amp_screener(&image, wsize, thresh, 1, &bright);
    CM_timer_stop(0);

    printf("# %d bright returns detected (out of %d).\n", nreturns,
           positionsof (current));
    printf ("%12.5f %12.5f %12.5f # amp %d %d %d %g ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), num_rows, num_cols, wsize,
            thresh, positionsof (Imageshape), positionsof (physical));

  }
}
```

B.6 Julia Set

```
/* Id: julia.cs,v 1.1 1996/01/09 20:40:52 pab Exp
 * Another example program from Justin R. Smith.  Modified for
 * correctness and appropriateness as a benchmark.  This version also
 * modified for load balance: rows use a cyclic decomposition so
 * adjacent rows appear on adjacent processors. */

#include <stdio.h>
#include <cm/timers.h>

shape [][] plane;

int main (int argc,
          char * argv [])
{
  int nrows;
  int ncols;
  int niters;
  float rmin, rmax, cmin, cmax;
  float p_r, p_c;

  nrows = ncols = 512;
  niters = 100;
  if (1 < argc) {
    nrows = atoi (argv [1]);
  }
  if (2 < argc) {
```

```
    ncols = atoi (argv [2]);
  }
  if (3 < argc) {
    niters = atoi (argv [3]);
  }
  rmin = cmin = -2.0;
  rmax = cmax = 2.0;
  p_r = 0.23;
  p_c = 0.13;

  allocate_shape (&plane, 2, nrows, ncols);
  with (plane) everywhere {
    float:plane r, c, r1;
    bool:plane injulia;
    int:current pivot;
    int i, j;

    /* Set up plane to range from -2..+2 */

    /* Set up pivots to be a permutation of pcoord (0) such that when
     * row R is on processor N, row R+1 is on processor N+1.  This
     * should improve load balance when whole rows are inactive */
    i = dimof (plane, 0) % dimof (physical, 0);
    j = dimof (plane, 0) / dimof (physical, 0);
    if (0 < i) {
      j++;
    }
    pivot = (pcoord (0) / dimof (physical, 0)) + j * (pcoord (0) %
                                                 dimof (physical, 0));
    if (0 < i) {
      where ((pcoord (0) % dimof (physical, 0)) > i) {
        pivot -= (pcoord (0) % dimof (physical, 0)) - i;
      }
    }

    r = rmin + (rmax - rmin) * pivot / dimof (current, 0);
    c = cmin + (cmax - cmin) * pcoord (1) / dimof (current, 1);

    /* Initially assume that all points are in the Julia set.  */
    injulia = 1;

    CM_timer_clear (0);
    CM_timer_start (0);

    /* Compute the first niters z's for each point of the selected
     * region of the complex plane. */
    for (i = 0; i < niters; i++) {
      /* We only work with points still thought to be in the Julia
       * set. */
```

```
      where (injulia) {
        r1 = r * r - c * c + p_r;
        c = 2.0 * r * c + p_c;
        r = r1;
        /* If sqr(|pt|) is less than 5, we're still active. */
        injulia &= (5.0 >= (r * r + c * c));
      }
    }
    CM_timer_stop (0);
    printf ("# %d of %d are active at end.\n", += injulia,
            positionsof (current));
    printf ("%12.5f %12.5f %12.5f # julia %d %d %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), nrows, ncols, niters,
            positionsof (current), positionsof (physical));
  }

  deallocate_shape (&plane);
}
```

B.7 Matrix Multiply

```
/* Id: mm.cs,v 1.1 1996/01/09 20:40:52 pab Exp
 * Basic matrix multiplication benchmark.
 */

#include <stdio.h>
#include <stdlib.h>
#include <cscomm.h>
#include <cm/timers.h>

shape [][] Matrix;

void
matmult (float:current * ma,
         float:current * mb,
         float:current * res)
{
  int col;

  everywhere {
    float:current mbt;

    [pcoord (1)][pcoord (0)] mbt = *mb;
    /* For each column of mb, spread it across, do an element-wise
     * multiply, and reduce into the answer */
    for (col = 0; col < dimof (current, 0); col++) {
```

```
      reduce (res, *ma * copy_spread (&mbt, 0, col), 1, CMC_combiner_add,
              col);
    }
  }
  return;
}

int
main (int argc, char * argv [])
{
  int order;

  order = 128;
  if (1 < argc) {
    order = atoi (argv [1]);
  }
  allocate_shape (&Matrix, 2, order, order);
  with (Matrix) everywhere {
    float:current mat;
    float:current m1;
    float:current m2;

    m1 = (prand () %% 1000) / 500.0;
    m2 = (prand () %% 1000) / 500.0;
    CM_timer_clear (0);
    CM_timer_start (0);
    matmult (&m1, &m2, &mat);
    CM_timer_stop (0);
    printf ("%12.5g %12.5g %12.5g # mm %d ; VP %d; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), order, positionsof (current),
            positionsof (physical));
  }
  deallocate_shape (&Matrix);
}
```

B.8 Rank Filter

```
/* Id: rf.cs,v 1.2 1996/01/22 15:38:38 pab Exp
 * Created: Tue Jun  6 13:36:36 1995
 * Peter A. Bigot (pab@clotho)
 * Last Revised:
 *
 * Description:
 * C* version of the rank filter benchmark.
 */

/* ------------ */
```

```
/* Include Files */

#include <stdio.h>              /* Standard input/output routines */
#include <stdlib.h>             /* Standard library routines */
#include <assert.h>             /* Debugging assertion macro */
#include <cscomm.h>
#include <cm/timers.h>
#include <string.h>

/* Rank filter: assign each pixel the ridx'th largest value in the win
 * x win window surrounding it. */
void
rank_filter (unsigned char:void * imp,
             unsigned char:void * outp,
             int win,
             int ridx)
{
  int wdnhalf = win / 2;
  int wuphalf = win - wdnhalf - 1;

  with (shapeof (*imp)) everywhere {
    unsigned char:current timage;
    unsigned char:current wval;
    unsigned char:current * rvals;
    unsigned char:current swaptmp;
    int dc;
    int i, j, k;

    /* Allocate a parallel array to hold the first ridx elements in
     * the window in sorted order.  Initialize it to the max value of
     * a pixel, so propagation occurs properly. */
    rvals = palloc (current, (ridx+1)*boolsizeof (unsigned char));
    memset (rvals, 255, (ridx+1)*boolsizeof (unsigned char));

    /* Start by shifting the image so that the upper left corner of
     * the window each pel is interested in is positioned at the
     * current position.  (We use torus so we don't have to worry
     * about shifting necessary values out of bounds during the walk,
     * then shifting back in the fills.)  Then we walk each row of the
     * window, and insert the value from that position in the window
     * into its sorted order in the rvals array.  Values that go past
     * the ridx'th element fall off the end.  The walk is
     * snake-row-major, rather than spiral, because this allows most
     * shifts to be along axis 1, which generally requires no
     * communication. */
    to_torus (&timage, imp, boolsizeof (*imp), wdnhalf, wdnhalf);
    dc = -1;
    for (i = 0; i < win; i++) {
      j = 0;
```

```
      while (j < win) {
        /* Propagate the value from this position into its proper
         * position in the sorted array. */
        wval = timage;
        k = 0;
        while (k <= ridx) {
          where (wval < rvals [k]) {
            swaptmp = rvals [k];
            rvals [k] = wval;
            wval = swaptmp;
          }
          k++;
        }

        /* Shift over one column if we're not at the edge */
        if (++j < win) {
          to_torus_dim (&timage, &timage, boolsizeof (timage), 1, dc);
        }
      }
      /* Shift up to the next row.  The next row shifts in the opposite
       * direction, to get the desired snaking effect. */
      if (i < win-1) {
        to_torus_dim (&timage, &timage, boolsizeof (timage), 0, -1);
      }
      dc = -dc;
    }
    /* Pull out the desired value from the sorted set. */
    *outp = rvals [ridx];
    pfree (rvals);
  }
}


int main (int argc,             /* Number of command line arguments */
          char * argv [])       /* Array of command line arguments */
{
  int nrows;            /* Rows in image */
  int ncols;            /* Columns in image */
  int window;           /* Window size */
  int rankid;           /* Desired statistic of window */
  shape Image;          /* Image shape */

  nrows = ncols = 32;
  window = 3;
  if (1 < argc) {
    nrows = atoi (argv [1]);
  }
  if (2 < argc) {
    ncols = atoi (argv [2]);
```

```
  }
  if (3 < argc) {
    window = atoi (argv [3]);
  }
  rankid = (window * window) / 2;
  if (4 < argc) {
    rankid = atoi (argv [4]);
  }

  allocate_shape (&Image, 2, nrows, ncols);
  with (Image) {
    unsigned char:current img;
    unsigned char:current mfimg;

    img = prand () %% 256;
    CM_timer_clear (0);
    CM_timer_start (0);
    rank_filter (&img, &mfimg, window, rankid);
    CM_timer_stop (0);
    printf ("%12.5g %12.5g %12.5g # rf %d %d %d %d ; VP %d ; P %d\n",
            CM_timer_read_elapsed (0), CM_timer_read_cm_busy (0),
            CM_timer_read_cm_idle (0), nrows, ncols, window, rankid,
            positionsof (current), positionsof (physical));
  }
  deallocate_shape (&Image);
  return (0);
}
```

REFERENCES

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., & Wagener, J. L. (1992). *Fortran 90 Handbook: Complete ANSI/ISO Reference*. McGraw Hill. Cited on pp. 3, 20, 184.

Adve, V., Carle, A., Granston, E., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Mellor-Crummey, J., Warren, S., & Tseng, C.-W. (1994). Requirements for data-parallel programming environments. *IEEE Parallel and Distributed Technology*, *2*(3). Cited on p. 2.

Agrawal, G., Sussman, A., & Saltz, J. (1993). Compiler and runtime support for structured and block structured applications. In *Supercomputing '93*, pp. 578–587. ACM Press. Cited on p. 122.

Agrawal, G., Sussman, A., & Saltz, J. (1995). An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, *6*(7), 747–754. Cited on p. 10.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. Cited on p. 35.

American National Standards Institute (1989). *American National Standard for Information Systems—Programming Language—C*. ANSI. Cited on pp. 3, 6, 8.

Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., & Townsend, G. (1988). An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, *10*(1), 51–86. Cited on p. 17.

Aspinall, R., & Veitch, N. (1993). Habitat mapping from satellite imagery and wildlife survey data using a Bayesian modeling procedure in a GIS. *Photogrammetric Engineering and Remote Sensing*, *59*(4), 537–543. Cited on p. 16.

Bailey, D. H. (1991). Twelve ways to fool the masses when giving performance results on parallel computers. Tech. rep. RNR-91-020, NASA Ames Research Center. Cited on p. 146.

Balasundaram, V., Fox, G., Kennedy, K., & Kremer, U. (1991). A static performance estimator to guide data partitioning decisions. In *Third Principles and Practice of Parallel Programming*, pp. 213–223. ACM. Cited on p. 63.

Banerjee, U. (1988). *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers. Cited on p. 2.

Barnett, M., Gupta, S., Payne, D. G., Shuler, L., Geijn, R. v., & Watts, J. (1994). Building a high-performance collective communication library. In *Supercomputing '94*, pp. 107–116. IEE Computer Society Press. Cited on p. 82.

Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., & Zagha, M. (1993). Implementation of a portable nested data-parallel language. In *Fourth Principles and Practice of Parallel Programming*, pp. 102–111. Cited on p. 21.

Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., & Wu, M.-Y. (1993). Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Supercomputing '93*. ACM Press. Cited on pp. 17, 126.

Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., Ranka, S., & Wu, M.-Y. (1994). Compiling Fortran 90D/HPF for distributed memory MIMD computers. *JPDC*, *21*, 15–26. Cited on pp. 4, 120.

Brezany, P., Gerndt, M., Sipkova, V., & Zima, H. P. (1992). SUPERB support for irregular scientific computations. In *Proceedings / Scalable High Performance Computing Conference, SHPCC-92*. IEEE Computer Society Press. Cited on p. 120.

Bruck, J., Dolev, D., Ho, C.-T., Rosu, M.-C., & Strong, R. (1994). Efficient message passing interface (MPI) for parallel computing on clusters of workstations. Tech. rep. RJ 9924 (87305) 12/13/94, IBM Research Division. Cited on p. 100.

Brustoloni, J. C., & Bershad, B. N. (1993). Simple protocol processing for high-bandwidth low-latency networking. Tech. rep. CMU-CS-93-132, Carnegie Mellon University. Cited on p. 61.

Budd, T. (1988). *An APL Compiler*. Springer–Verlag. Cited on p. 57.

Chandranmenon, G. P., Russell, R. D., & Hatcher, P. J. (1994). Providing an execution environment for C* programs on a Mach-based PC cluster. Tech. rep. TR 94-20, University of New Hampshire. Cited on p. 65.

Chang, C.-H., Flower, D., Forecast, J., Gray, H., Hawe, B., Nadkarni, A., Ramakrishnan, K., Shikarpur, U., & Wilde, K. (1994). High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP. In *Third High Performance Distributed Computing*, pp. 35–42. IEEE. Cited on p. 61.

Chapman, B., Zima, H., & Mehrotra, P. (1994). Extending HPF for advanced data-parallel applications. *IEEE Parallel and Distributed Technology*, *2*(3). Cited on pp. 120, 126.

Chatterjee, S., Gilbert, J. R., Long, F. J., Schreiber, R., & Teng, S.-H. (1993). Generating local addresses and communication sets for data-parallel programming. In *4th Principles and Practive of Parallel Programming*, pp. 149–158. ACM Press. Cited on p. 35.

Cheung, A. L., & Reeves, A. P. (1992). High performance computing on a cluster of workstations. In *First High Performance Distributed Computing*, pp. 152–160. IEEE Computer Society Press. Cited on p. 16.

Ching, W.-M., & Ju, D. (1991). Execution of automatically parallelized APL programs on RP3. *IBM Journal of Research and Development*, *35*(5/6). Cited on p. 21.

Ching, W.-M., & Katz, A. (1994). An experimental APL compiler for a distributed memory parallel machine. In *Proceedings of Supercomputing '94*. Cited on p. 21.

Choudhary, A., Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Ranka, S., & Tseng, C.-W. (1993). Unified compuation of Fortran 77D and 90D. *ACM Letters on Programming Languages and Systems*, *2*(1-4), 95–114. Cited on pp. 17, 18, 20.

Congalton, R. G., Green, K., & Teply, J. (1993). Mapping old growth forests on national forest and park lands in the Pacific Northwest from remotely sensed data. *Photogrammetric Engineering and Remote Sensing*, *49*(4), 529–535. Cited on p. 16.

Crandall, P. E., & Quinn, M. J. (1993). Block data decomposition for data-parallel programming on a heterogeneous workstation network. In *2nd Intl. Symp. High Performance Distributed Computing*, pp. 42–49. IEEE Computer Society Press. Cited on pp. 20, 56, 65.

Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., von Eicken, T., & Yelick, K. (1993). Parallel programming in Split-C. In *Supercomputing '93*, pp. 262–273. ACM Press. Cited on p. 21.

Das, R., Ponnusamy, R., Saltz, J., & Mavriplis, D. (1992). Distributed memory compiler methods for irregular problems—data copy reuse and runtime partitioning. In Saltz, J., & Mehrotra, P. (Eds.), *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, pp. 185–219. Elsevier Science Publishers. Cited on pp. 122, 125.

Das, R., Uysal, M., Saltz, J., & Hwang, Y.-S. (1994). Communication optimizations for irregular scientific computations on distributed memory architectures. *JPDC*, *22*, 462–478. Cited on p. 120.

Dongarra, J., van de Geijn, R., & Walker, D. (1992). A look at scalable dense linear algebra libraries. In *Scalable High Performance Computing Conference–92*, pp. 372–379. IEEE Computer Society Press. Cited on p. 26.

Druschel, P. (1994). *Operating System Support for High-Speed Networking*. TR 94-24, The University of Arizona. Cited on pp. 61, 100.

Fallah-Adl, H., JáJá, J., Liang, S., Kaufman, Y. J., & Townshend, J. (1995). Efficient algorithms for atmospheric correction of remotely sensed data. Tech. rep. UMD CS-TR-3464, UMIACS, University of Maryland. Cited on p. 21.

Fischer, C. N., & LeBlanc, Jr., R. J. (1988). *Crafting a Compiler*. Benjamin/Cummings. Cited on p. 35.

FORE Systems (1994). *ForeRunner SBA-100/-200 ATM SBus Adapter User's Manual, Revision Level D*. Cited on p. 63.

Fox, G. (1988). What have we learnt from using real parallel machines to solve real problems. In *Third Conference on Hypercube Concurrent Computers and Applications*, Vol. 2, pp. 897–955. ACM Press. Cited on p. 3.

Frankel, J. L. (1991). A reference description of the C* language. Tech. rep. TR-253, Thinking Machines Corporation. Cited on pp. 3, 8.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., & Sunderam, V. (1994). PVM 3 user's guide and reference manual. Tech. rep. ORNL/TM-12187, Oak Ridge National Laboratory. Cited on p. 59.

Gilman, L., & Rose, A. J. (1984). *APL: An Interactive Approach* (3rd edition). John Wiley and Sons. Cited on p. 21.

Green, K., Kempka, D., & Lackey, L. (1994). Using remote sensing to detect and monitor land-cover and land-use change. *Photogrammetric Engineering and Remote Sensing*, *60*(3), 331–337. Cited on p. 16.

Hamey, L. G., Webb, J. A., & Wu, I.-C. (1989). An architecture independent programming language for low-level vision. *Computer Vision, Graphics, and Image Processing*, *48*, 246–264. Cited on p. 21.

Harris, J., Bircsak, J. A., Bolduc, M. R., Diewald, J. A., Gale, I., Johnson, N. W., Lee, S., Nelson, C. A., & Offner, C. D. (1995). Compiling high performance fortran for distributed-memory systems. *Digital Technical Journal*, *7*(3). Cited on p. 2.

Hatcher, P. J., & Quinn, M. J. (1991). *Data-Parallel Programming on MIMD Computers*. MIT Press. Cited on pp. 16, 17, 20.

Hennessy, J. L., & Patterson, D. A. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann. Cited on pp. 2, 46.

High Performance Fortran Forum (1993). *High Performance Fortran Language Specification* (1.0 edition). Cited on pp. 3, 20, 24, 184.

Hillis, W. D., & Steele Jr., G. L. (1986). Data parallel algorithms. *Communications of the ACM*, *29*(12), 1170–1183. Cited on p. 3.

Hillis, W. D., & Tucker, L. W. (1993). The CM-5 Connection Machine: A scalable supercomputer. *Communications of the ACM*, *36*(11), 30–40. Cited on p. 146.

Hiranandani, S., Kennedy, K., & Tseng, C.-W. (1993). Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*, pp. 338–350. ACM Press. Cited on pp. 10, 20.

Hiranandani, S., Kennedy, K., & Tseng, C.-W. (1994). Evaluating compiler optimizations for Fortran D. *JPDC*, *21*, 27–45. Cited on pp. 63, 104.

Knobe, K., Lukas, J. D., & Guy L. Steele, J. (1990). Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, *8*, 102–119. Cited on p. 23.

Knuth, D. E. (1973). *Sorting and Searching*, Vol. 3 of *The Art of Computer Programming*. Addison-Wesley. Cited on pp. 112, 117.

Koelbel, C. (1990). *Compiling Programs for Nonshared Memory Machines*. Csd-tr-1037, Purdue University. Cited on pp. 10, 139.

Koelbel, C., & Mehrotra, P. (1991). Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, *2*(4), 440–451. Cited on pp. 21, 119, 122.

Koelbel, C., Mehrotra, P., & Rosendale, J. V. (1990). Supporting shared data structures on distributed memory architectures. In *2nd Principles and Practice of Parallel Programming*. ACM Press. Cited on p. 122.

Lapadula, A. J., & Herold, K. P. (1994). A retargetable C* compiler and run-time library for mesh-connected MIMD multicomputers. Tech. rep. TR 92-15, University of New Hampshire. Cited on pp. 16, 20, 38, 39, 40, 120, 141, 178.

LaRosa, J. A. (1995). A protocol for network-based concurrent computing.. Cited on p. 84.

Mahéo, Y., & Pazat, J.-L. (1993). Distributed array management for HPF compilers. Publication interne 787, Institut de Recherche en Informatique et Systemes Aleatoires. Cited on p. 23.

Mason, J. R., Hatcher, P. J., & Chappelow, S. (1994). Optimizing irregular communication patterns in UNH C*. Tech. rep. TR 94-14, University of New Hampshire. Cited on pp. 20, 120, 125, 178.

Mellor-Crummey, J. M., & Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *TOPLAS*, *9*(1), 21–65. Cited on p. 79.

Message Passing Interface Forum (1994). *MPI: A Message-Passing Interface Standard*. Cited on p. 59.

Mitra, P., Payne, D. G., Shuler, L., van de Geijn, R., & Watts, J. (1995). Fast collective communication libraries, please. Tech. rep. TR95-22, University of Texas at Austin. Cited on pp. 59, 77.

Mosberger, D., Peterson, L. L., & O'Malley, S. (1995). Protocol latency: MIPS and reality. Tech. rep. TR 95-02, University of Arizona. Cited on p. 71.

Mosberger, D., Turner, C. J., & Peterson, L. L. (1994). Exploiting highly reliable networks with careful protocols. Tech. rep. TR 94-14, University of Arizona. Cited on pp. 65, 101.

Numerical C Extensions Group of X3J11 (1994). Data parallel C extensions. Tech. rep. X3J11/94-080 / WG13/N395, ANSI X3J11. Cited on pp. 3, 8, 21, 184.

Oed, W. (1993). The Cray Research massively parallel processor system: CRAY T3D.. Cited on p. 60.

Papadopoulos, C., & Parulkar, G. M. (1993). Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation. *IEEE/ACM Transactions on Networking*, *1*(2), 199–216. Cited on p. 73.

Ponnusamy, R., Hwang, Y.-S., Das, R., Saltz, J. H., Choudhary, A., & Fox, G. (1995). Supporting irregular distributions using data-parallel languages. *IEEE Parallel and Distributed Technology*, *3*(1), 12–24. Cited on pp. 120, 121, 122, 126.

Ponnusamy, R., Saltz, J., & Choudhary, A. (1993). Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing '93*, pp. 361–370. ACM Press. Cited on p. 122.

Ponnusamy, R., Saltz, J., Choudhary, A., Hwang, Y.-S., & Fox, G. (1995). Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, *6*(8), 815–831. Cited on p. 120.

Press, W. H., Flannery, B. P., Teukoloky, S. A., & Vetterling, W. T. (1984). *Numerical Recipes in C.* Cambridge University Press. Cited on p. 49.

Pure Software (1994). *Purify User's Guide*. Cited on p. 39.

Richards, J. A. (1994). *Remote Sensing Digital Image Analysis* (2nd edition). Springer-Verlag. Cited on pp. 4, 16, 41.

Rose, J. R., & Steele Jr., G. L. (1987). C*: An extended C language for data parallel programming. In *Proceedings 2nd International Conference on Supercomputing*, Vol. 2, pp. 2–16. International Supercomputing Institute. Cited on p. 20.

Rosing, M., Schnabel, R. B., & Weaver, R. P. (1991). The DINO parallel programming language. *JPDC*, *13*, 30–42. Cited on p. 21.

Sahni, S., & Thanvantri, V. (1996). Performance metrics: Keeping the focus on runtime. *IEEE Parallel & Distributed Technology*, *4*, 43–56. Cited on p. 146.

Sharma, S. D., Ponnusamy, R., Moon, B., shin Hwang, Y., Das, R., & Saltz, J. (1994). Runtime and compile-time support for adaptive irregular problems. In *Supercomputing '94*, pp. 97–106. IEE Computer Society Press. Cited on p. 120.

Sheffler, T. J., Schreiber, R., Gilbert, J. R., & Chatterjee, S. (1994). Aligning parallel arrays to reduce communication. Tech. rep. RIACS TR 94.10, Research Institute for Advanced Computer Science, NASA Ames. Cited on p. 23.

Skjellum, A. (1993). Scalable libraries in a heterogeneous environment. In *Second High Performance Distributed Computing*, pp. 13–20. IEEE. Cited on p. 65.

Socha, D. G. (1991). *Supporting fine-grain computation on distributed memory parallel computers*. TR 91-07-01, University of Washington. Cited on p. 56.

Thinking Machines Corporation (1993). *C* Programming Guide*. Thinking Machines Corporation. Cited on pp. 8, 122, 173.

Tseng, C.-W. (1993). *An Optimizing Fortran-D Compiler for MIMD Distributed-Memory Machines*. Rice COMP TR93–199, Rice University. Cited on pp. 4, 20, 24, 56, 129.

Turner, C. J., & Turner, J. G. (1994). Adaptive data parallel methods for ecosystem monitoring. In *Supercomputing '94*, pp. 281–290. IEEE Computer Society Press. Cited on pp. 4, 16, 41.

Turner, C. J. (1994). *Cluster-C*: A Data Parallel Computing Architecture for Automated Remote-Sensing Applications*. Ph.D. thesis, The University of Arizona. Cited on pp. 50, 61, 101.

von Eicken, T., Culler, D. E., Goldstein, S. C., & Schauser, K. E. (1992). Active messages: A mechanism for integrated communication computation. Tech. rep. UCB/CSD 92/#675, University of California, Berkeley. Cited on p. 74.

Voorhees, H., & Tucker, L. W. (1992). Efficient representation and transformation of image data on the connection machine system.. *Machine Vision and Applications*, *5*(2), 63–83. Cited on p. 41.

Weissman, J. B., & Grimshaw, A. S. (1994). Network partitioning of data parallel computations. In *3rd Intl. Symp. High Performance Distributed Computing*, pp. 140–156. IEEE Computer Society Press. Cited on p. 65.

INDEX