# Automatic Translation of Data Parallel Programs to Message Passing Programs

**T. Brandes**

Internal Report No. Adaptor 93-1

January 29, 1993

*High Performance Computing Center*
*German National Research Institute for Computer Science*
*P. O. Box 1316*
*D-5205 Sankt Augustin 1*
*Federal Republic of Germany*
*Tel.: +49 (0)2241 / 14-2492*
*E-mail: brandes@gmd.de*

# Automatic Translation of Data Parallel Programs to Message Passing Programs

**T. Brandes**

*German National Research Center for Computer Science,*
*P.O. Box 1316, D-5205 Sankt Augustin 1, FRG*

**Abstract**

Data parallel programming stands for single threaded, global name space, and loosely synchronous parallel computation. This kind of parallel programming has been proven to be very user-friendly, easy to debug and easy to use. But this programming model is not available for most message passing multiprocessor architectures.

Adaptor (Automatic Data Parallelism Translator) is a tool that transforms data parallel programs written in Fortran with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing. The current version supports especially the translation of Connection Machine Fortran programs to message passing programs.

Adaptor is not a compiler but a source to source transformation that generates Fortran 77 host and node programs with message passing. The new generated source codes can run on most parallel architectures. By this way it is possible to write data parallel programs that are portable for a wide range of parallel machines.

In the following the realization of such a translation system is described. It will be shown how efficient this approach is and what kind of optimizations could be useful for compiling data parallel programs.

# 1 Introduction

MIMD (multiple instruction, multiple data) architectures with distributed memory are the kind of parallel machines that are scalable and can be used for a wide range of scientific applications. Usually, these architectures are programmed with explicit message passing between the processes running on the different processors. As the message passing programming model is very error prune and difficult to use, many efforts have been made to offer other programming models that are easier to use.

The High Performance Fortran Forum has defined language extensions and modifications for Fortran to overcome these difficulties by supporting data parallel programming [Koe92]. This kind of programming can be defined as single threaded, global name space, and loosely synchronous parallel computation. The new language allows code tuning for various architectures and should guarantee top performance on MIMD and SIMD (single instruction, multiple data) computers with non-uniform memory access costs.

Many large scientific applications are expected to be programmed in this data parallel model (Fortran 77 with array extensions or Fortran 90). The parallelization tool Adaptor (Automatic Data Parallelism Translator) makes it possible to translate these programs to message passing programs already now. It transforms data parallel programs written in Fortran 77 with array extensions, parallel loops, and layout directives to parallel programs with explicit message passing.

Therefore the code with global data references together with a user specified or implicitly defined data distribution is translated into a program with local and non-local references, where the latter are satisfied by automatically inserting message-passing statements.

Experiments with many sequential programs and their Fortran 90 counterparts have shown [Fox91] that automatic methods could not parallelize the sequential version where it is possible for the version with explicit array operations. In Adaptor only the inherent parallelism of the array operations is used. Local array operations will be distributed among the available nodes, for non-local array operations efficient communication is generated as these operations have mostly regular communication patterns (e.g. global reductions, shift and spread operations).

In the following sections the functionality and the realization of the tool Adaptor is described. First results are presented and useful optimization issues will be discussed.

# 2 Translation of Data Parallel Programs

A prototype version of the Adaptor tool has been realized that transforms Fortran 77 or Fortran 90 programs with explicit data parallelism into parallel programs for MIMD architectures with explicit message passing. Though during the last months the High Performance Fortran Forum has defined language extensions for data parallelism, the current version has been designed especially to translate CM Fortran [Cor90] programs to message passing programs.

## 2.1 Properties of Adaptor

Though the user will need to understand some issues of parallelism and has to know for efficiency reasons where message passing will be generated, the effectiveness of Adaptor is based on the fact that the user has not to know any message passing command and not to manage the control of the data partitioning. He can change types of variables (e.g. single to double precision) and data distributions without rewriting any other statement in his program. He has not to write two versions of code (host and node program) and many global array operations are translated to the most efficient code for the underlying architecture.

The parallel program can be written in such a way that it can be developed on a serial machine and is also suitable for vector machines or parallel machines with shared memory. Many features supported by Adaptor result also in good execution times for these architectures. By this way, it helps to design programs that run efficiently on nearly all architectures.

The generated code of Adaptor should be as efficient as possible and competitive to a hand-coded Fortran program with message passing. Otherwise the acceptance of such a tool cannot be expected.

Adaptor supports the development of parallel codes that scale with the number of processors. No support is given for any kind of programming where the number of processors is fixed in any way. It makes heavy use of dynamic arrays and the executable version of the generated program can run for any number of processors without any recompilation.

### 2.2 Related Work

The data parallel programming style has been proven as to be user-friendly and easy to use. Many other systems have been developed during the last time that also support SIMD programming for MIMD architectures. A SIMD program is translated into an equivalent SPMD program (single program, multiple data stream). This has been done for C* [HLJ+91] or for Fortran 90 with additional layout directives [Mer91, WF91]. Though the latter systems are very similar to Adaptor there is no information about the efficiency of the generated message passing programs and about their availability.

Due to the introduction of High Performance Fortran, many compilers will be available in the next future and compiler optimizations are goals of some other projects [HKT91].

Further developments have been made to support data parallel programming in an object-oriented language like C++ [CCRS91]. This approach has the great advantage that no additional preprocessor or compiler is necessary. But due to the lack of efficiency there is not a great acceptance for scientific applications until now.

## 3 Overview of Adaptor

The Adaptor system consists of an interactive source-to-source transformation (XAdaptor), a library of routines for message passing and controlling array distributions (DALIB).

### 3.1 The Input Language

The input language of Adaptor can be defined as Fortran 77 with some restrictions (see section 4.1), but with many extensions like dynamic arrays, array operations, parallel loops and layout directives [Bra92].

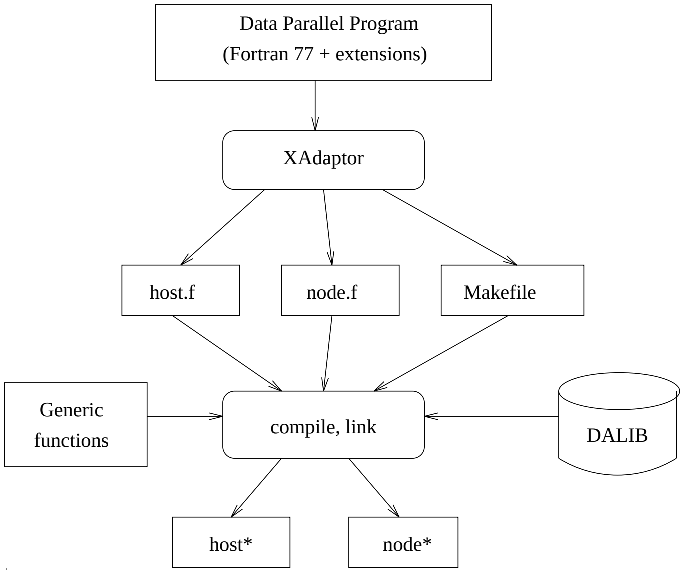


Figure 1: Overview of Adaptor

The central idea of an automatic adaptation is to distribute most of the given arrays among the available processors. This is done in such a way that most operations can be done locally without any need of communication. Where global operations are necessary the corresponding message passing statements are inserted automatically. Adaptor takes only advantage of the parallelism in the array operations and of the parallel loops. It has no features for automatic parallelization.

For the specification of data layouts in Adaptor similar directives as in CM Fortran are used. The user can define host arrays, replicated arrays and distributed arrays. In contrary to many other systems [Mer91, Cor91, FHK+91, Ger89] Adaptor supports only block distributions along the last dimension.

Alignment is a feature that can be used to reduce communication [KLS91] especially for a given program. It is supported in CM Fortran and in High Performance Fortran. For Adaptor it is not supported until now as the best alignment is done by declaring arrays with the same shape and the same layout when writing new parallel programs.

## 3.2 Generated Programs

For a given data parallel program Adaptor generates a host and a node program (hostnode model) or only a node program (hostless model).

The host program is running on the front end and the node program runs independently on all processing nodes. While the host program takes care about the host arrays and all I/O activities, the nodes are operating on the distributed arrays. Control flow, scalar variables and replicated arrays are replicated for the host process and all node processes.

## 3.3 Interactive Source-to-Source Transformation

The translation of the input file can be done as a batch job, but an interactive translation is also possible. A graphical environment allows the user to select units of the source program (program, functions, subroutines) or variables in a unit to get information about them (see figure 2).

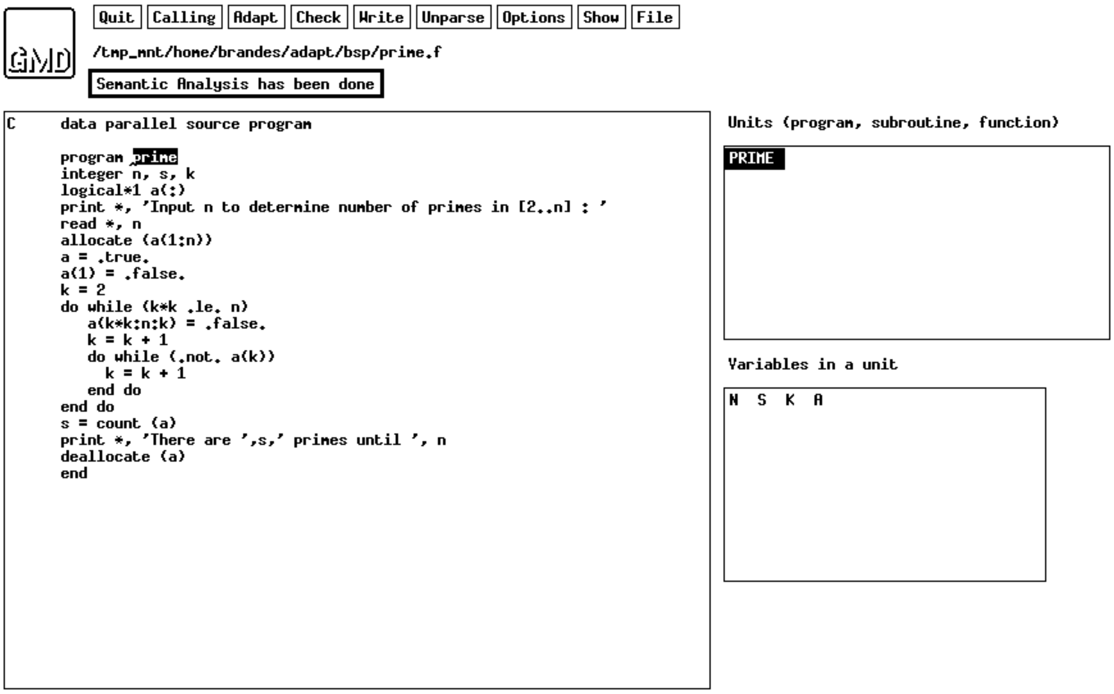


Figure 2: XAdaptor: Interactive Source-to-Source Transformation

Except the graphical interface, the whole source-to-source transformation of Adaptor is generated with a toolbox for compiler construction [GE90]. These tools have a great flexibility and can generate very efficient code. As intermediate language, abstract syntax trees will be used where the program module that defines the structure of the abstract syntax trees and provides general tree manipulating procedures is also generated by a tool.

For the analysis and transformation components the new compiler tool *Puma* is utilized [Gro91]. This tool cooperates with the generator for abstract syntax trees and supports the transformation and attribution of attributed trees. It is based on pattern-matching, unification and recursion. The flexibility of this tool allows not only to have a modular design but also to extend it in a way as one would expect from a knowledge-based system [BS87].

The graphical environment is realized with Athena widgets based on the X Window System [O'R90].

## 3.4 Distributed Array Library

For the realization of the communication needed for global operations on distributed arrays, many library functions will be used that build the DALIB (distributed array library). This library contains

- low level communication (send, receive, wait, ...),
- high level communication (broadcast, reduction, barrier, ...),
- functions to control the data partitioning,
- primitives for gathering and scattering data,
- timing functions and tracing facilities,
- parallel random number generator,
- and a X-Windows interface.

The DALIB is implemented in C. Most part of this library is portable between the different machines. Only the low level message passing commands, the timing functions and the random number generator have to be adapted to the hardware architecture.

Though the realization of the high level communication routines is based on the low level routines, these functions should be tuned for the underlying hardware architecture. As e.g. the CM 5 has an own control network, broadcasts and reductions are more efficient when using this network than using the data network by message passing.

One version of the DALIB is implemented upon the public domain software PVM [Sun90]. It guarantees the portability of the generated parallel programs to all machines where PVM is running. Another version exists for shared memory and virtual shared memory systems where the message passing is realized very efficiently via a shared memory segment.

At the moment the DALIB has been implemented for iPSC/860, net of SUN or IBM workstations, Alliant FX/2800, Parsytec GCel, CM5 and KSR 1.

## 3.5 Visualization of the Run Time Behavior

If the final parallel program is started with the trace flag switched on, a tracefile will be generated that gives information about the behavior of the parallel program. The information of the tracefile can be visualized and animated with the public domain software ParaGraph [HE91]. Especially the information about the utilization and communication can be used for further optimizations.

### 3.6 Availability

The source files of Adaptor, documentation files in PostScript and a number of example programs are available via 'anonymous ftp' from:

```
ftp.gmd.de        (129.26.8.90)
in subdirectory  gmd/adaptor
```

Currently version 0.1 is available, version 1.0 with more functionality, stability and more supported features will be available spring 1993.

# 4 Description of the Translation

The following steps are done during the source to source transformation of Adaptor:

1. The source program is parsed and an abstract syntax tree will be generated.
2. Symbol tables are created and used for a semantic analysis.
3. The real translation on the internal abstract syntax tree and symbol tables has four phases:
   (a) analysis to verify that the program is suitable for translation with Adaptor,
   (b) splitting up statements in local and non-local operations and creation of the necessary temporary variables,
   (c) initial translation and serialization,
   (d) final translation with generation of calls to the DALIB
4. The new internal abstract syntax tree is unparsed back to source text.

In the following the four phases of the real translation on the internal abstract syntax tree are described in more detail.

## 4.1 The Analysis Phase

In the first phase most checks are made to verify whether the source program can be translated with Adaptor. The current version has the following restrictions:

- no EQUIVALENCE or SAVE statements for distributed arrays,
- the use of distributed arrays in a DATA statement is not possible,

- no input/output statements or calls of external subroutines with distributed arrays (use replicated or host arrays and send the values to the nodes by using a primitive array assignment),
- arrays in COMMON blocks cannot be distributed (make them to local arrays of the main program and use a new parameter for subroutine calls),
- arrays in subroutine calls must have the same distribution as the dummy argument (this restriction can be observed by using temporary arrays and primitive array assignments),
- no parameter subroutines or functions,
- no return statement (replace it if necessary with a jump to the end of the function or subroutine),
- assumed shaped arrays are not allowed, assumed sized arrays cannot be distributed and not be used in array expressions,
- some restrictions for the use of intrinsic functions (in most cases it is possible to use parallel loops),
- correct layout of arrays used as parameters of intrinsic subroutines.

Many restrictions are only preliminary and will not exist in the next version. On the other side experiences have shown that the required changes in the data parallel source program help to think about more data parallelism and better data locality.

## 4.2 Temporary Variables

In many cases array assignments need communication. In this case Adaptor tries to split up the assignment in primitive array assignments with communication and local array assignments. Sometimes new temporary arrays have to be created.

```
      real A(N), B(N), C(N)     ! all distributed by default

      A(1:K) = A(1:K) * B(1:K) - C(K+1:N-K)

c     will result in

      real A(N), B(N), C(N), A_TMP1(N)

      A_TMP1(1:K) = C(K+1:N-K)             ! requires communication
      A(1:K) = A(1:K) * B(1:K) - A(1:K)    ! local operation
```

In many other situations, especially when using complex array operations, temporary variables will also be necessary. Due to the memory overhead attention should be paid by the user itself that not too many arrays will be created.

## 4.3 Initial Translations

In this phase static and automatic arrays that are distributed become allocatable arrays. By this way it will be possible that the generated message passing programs can run for different numbers of processors.

Local array operations will be translated to parallel loops, the forall statement will be translated to equivalent do loops. After this phase only parallel loops without communication will exist.

The following program part

```
integer N
parameter (N=100)
real A(N), B(N)
a = b
forall (i=2:n-1)
  a(i) = b(i) + a(i+1)
end forall
```

will be translated to:

```
INTEGER*4 N
PARAMETER ( N = 100 )
REAL*4 A (:), B (:), A_TMP1 (:)
INTEGER*4 I_1
ALLOCATE (A(1:N), B(1:N), A_TMP1(1:N))
DO I_1=1,N  !parallel
   A(I_1) = B(I_1)
END DO
A_TMP1(2:N-1) = A(3:N)    ! array movement
DO I=2,N-1  !parallel
   A(I) = B(I)+A_TMP1(I)
END DO
DEALLOCATE (A_TMP1, B, A)
```

## 4.4 Final Translation

Now all statements are classified whether they are local statements or communication statements. After the final translations it is guaranteed that

- a parallel loop is restricted to the part of the arrays that is owned by the node process,
- communication statements or movements are translated to corresponding calls of subroutines of the DALIB.

This small source code

```
real A(N), S
S = sum (A)
```

is translated to the following code:

```
INTEGER*4 A_DSP
INTEGER*4 A_LOW, A_HIGH
INTEGER*4 A_START, A_STOP, A_INC
REAL*4 A (:)
REAL*4 S
INTEGER*4 I_1
call dalib_define_array1 (A_DSP,4,1,N)
call dalib_array_pardim (A_DSP,A_LOW,A_HIGH)
ALLOCATE (A(A_LOW:A_HIGH))
S = 0.0
DO I_1=A_LOW,A_HIGH
   S = S+A(I_1)
END DO
call dalib_real_sum (S)
call dalib_undefine_array (A_DSP)
DEALLOCATE (A)
```

The generated program alternates between phases of local computations and more synchronous phases of local and global communication.

# 5 Experiments and Results

With the current system it is possible to translate CM Fortran programs to message passing programs. Due to the restrictions in the distribution of parallel variables and to some other restrictions it is necessary to make some changes in the given CM Fortran programs.

## 5.1 The Purdue Set

For testing the first version of Adaptor the High Performance Fortran Benchmark Suite has been utilized [MFL+92] where many data parallel programs are given in different versions.

The Purdue set (J.R. Rice set) with 14 simple data parallel problems has been used to test the efficiency of the generated message passing programs.

Three different versions of the programs have been considered:

- the Fortran 77 code can be translated for one node and is used to measure real speed ups,
- the CM Fortran version gives results for the Connection Machine and is used for Adaptor with slightly changes to get automatically generated message passing programs for different parallel machines,
- the parallel version, Fortran 77 with explicit message passing based on PICL, is used to compare the results of Adaptor with a hand-coded message passing program.

| no | short description of the problem | problem size |
|---|---|---|
| 1 | Trapezoidal rule | 1048576 |
| 2 | reduction function 1 | 1024 x 1024 |
| 3 | reduction function 2 | 1024 x 1024 |
| 4 | reduction function 3 | 524288 |
| 5 | simple search | 128 x 4096 |
| 6 | tridiagonal set of lin. equations | 65536 |
| 7 | Lagrange interpolation | 10 x 32768 |
| 8 | divided differences | 65536 x 8 |
| 9 | finite differences | 512 x 512 |
| 11 | Fourier's moments | 262144 |
| 12 | array's construction | 1023 x 511 |
| 13 | floating point arithmetic | 262144 |
| 14 | Simpson's and Gauss' integration | 262144 |
| 15 | Chebyshev interpolation | 16384 |

## 5.2 Sequential and Parallel Version

The following table shows the results of the sequential Fortran 77 version running on one node of the iPSC/860 compared with the generated message passing program only running on one node. The speed up or better slow down of the parallel version with the sequential version is given.

| no | problem size | time of sequential F77 version (1 node) | Adaptor version (1 node) | speed up |
|---|---|---|---|---|
| 1 | 1048576 | 284.7 s | 290.6 s | 0.98 |
| 2 | 1024 x 1024 | 45.9 s | 33.8 s | 1.36 |
| 3 | 1024 x 1024 | 47.0 s | 10.3 s | 4.56 |
| 4 | 524288 | 222.1 s | 185.5 s | 1.20 |
| 5 | 128 x 4096 | 175.9 s | 209.3 s | 0.84 |
| 6 | 262144 | 414.7 s | 1008.9 s | 0.41 |
| 7 | 10 x 32768 | 131.1 s | 143.4 s | 0.91 |
| 8 | 65536 x 8 | 104.8 s | 186.5 s | 0.56 |
| 9 | 512 x 512 | 46.2 s | 46.2 s | 1.00 |
| 11 | 262144 | 289.7 s | 289.1 s | 1.00 |
| 12 | 1023 x 511 | 182.6 s | 21.9 s | 8.34 |
| 13 | 262144 | 588.4 s | 555.9 s | 1.06 |
| 14 | 262144 | 35.5 s | 25.2 s | 1.41 |
| 15 | 16384 | 133.2 s | 92.0 s | 1.45 |

The results show that the Adaptor version is faster than the F77 counterpart for the problems 2, 3, 4, 12, 14 and 15. The following reasons are responsible for this effect:

- In problem 2, 3 and 12 Adaptor generates a different loop nesting (innermost loop is always for the first index that results in stride 1 for the loop iterations). After loop interchanging and loop distribution the sequential F77 version was as fast as the Adaptor version.
- In problem 4 only one loop fusion makes the F77 version as fast the generated one.
- In problem 14 and 15 the Adaptor version has a vector version of the function that is integrated. By this way there is only one subroutine call instead of a call for every point.

But of course for most programs the Adaptor version is a little bit slower due to the generated communication and due to additional memory movements. Especially the problems 6 and 8 require much communication.

## 5.3 Efficiency and Scalability

The following table shows the speed ups and efficiencies for the generated message passing programs on the iPSC/860 for 8, 16 and 32 nodes.

| no | Size | iPSC(8) | iPSC(16) | iPSC(32) |
|---|---|---|---|---|
| 1 | 1048576 | 7.96 (99.5 %) | 15.79 (98.7 %) | 31.25 (97.6 %) |
| 2 | 1024 x 1024 | 7.41 (92.6 %) | 14.08 (88.0 %) | 28.17 (88.0 %) |
| 3 | 1024 x 1024 | 6.44 (80.5 %) | 12.88 (80.5 %) | 25.75 (80.5 %) |
| 4 | 524288 | 6.08 (76.0 %) | 13.16 (82.2 %) | 26.13 (81.6 %) |
| 5 | 128 x 4092 | 7.12 (89.0 %) | 14.14 (88.4 %) | 27.91 (87.2 %) |
| 6 | 262144 | 4.86 (60.7 %) | 8.75 (54.7 %) | 16.17 (50.5 %) |
| 7 | 10 x 32768 | 7.97 (99.6 %) | 15.76 (98.5 %) | 31.17 (97.4 %) |
| 8 | 65536 x 8 | 7.61 (95.2 %) | 13.92 (87.0 %) | 25.90 (80.9 %) |
| 9 | 512 x 512 | 7.97 (99.6 %) | 15.40 (96.3 %) | 28.88 (90.2 %) |
| 11 | 262144 | 7.94 (99.3 %) | 14.90 (93.1 %) | 29.20 (91.3 %) |
| 12 | 1023 x 511 | 6.84 (85.5 %) | 10.95 (68.4 %) | 16.85 (52.6 %) |
| 13 | 262144 | 7.91 (98.9 %) | 15.80 (98.7 %) | 31.71 (99.0 %) |
| 14 | 262144 | 7.20 (90.0 %) | 14.00 (87.5 %) | 25.20 (78.8 %) |
| 15 | 16384 | 7.48 (93.5 %) | 14.84 (92.7 %) | 28.75 (89.8 %) |

These results verify that the scalability of the data parallel programs results also in scalability of the message passing programs.

## 5.4 Adaptor vs. hand-coded message passing programs

The parallel programs based on PICL stand for portable hand-coded message passing programs. The most interesting results came up when comparing these programs with the automatically generated message passing programs of Adaptor.

Both versions are portable parallel programs. Both versions are able to run on different number of processors. The hand-coded version realizes this by having the whole data structure replicated on all arrays but every node works only on a subset of the arrays. This has the disadvantage that for bigger problems the code has to be rewritten completely. The Adaptor version can also be used for bigger problems with the only restriction that the program will not run on smaller machines size.

- A hand-coded message passing program of problem 6 was not available,
- for problem 2, 3, 4, 14 Adaptor was much more faster,
- in all other problems a little bit faster or nearly the same (problems 5, 8, 11, 12, 13).

| no | Size | nodes | F77 + PICL | Adaptor |
|---|---|---|---|---|
| 1 | 1048576 | 16 | 25.4 s | 18.4 s |
| 2 | 1024 x 1024 | 16 | 5.2 s | 2.4 s |
| 3 | 1024 x 1024 | 16 | 7.1 s | 0.8 s |
| 4 | 524288 | 16 | 24.3 s | 14.1 s |
| 7 | 10 x 32768 | 16 | 9.0 s | 9.1 s |
| 9 | 512 x 512 | 16 | 5.3 s | 3.0 s |
| 14 | 262144 | 16 | 11.6 s | 1.8 s |
| 15 | 16384 | 16 | 14.6 s | 6.2 s |

As a hand-coded program should always be faster than an automatically generated message passing program, the results show in any case that Adaptor can generate more efficient programs than just straightforward hand written message passing programs.

## 5.5 Full vs. Loosely Synchronous Execution

The last table compares the results of the programs running on the SIMD Connection Machine CM 2 and on the MIMD iPSC/860.

| no | Size | iPSC/860 (16 nodes) | iPSC/860 (32 nodes) | CM2 8k | CM2 16k |
|---|---|---|---|---|---|
| 1 | 1048576 | 18.4 s | 9.3 s | 9.2 s | 4.5 s |
| 2 | 1024 x 1024 | 2.4 s | 1.2 s | 4.2 s | 2.2 s |
| 3 | 1024 x 1024 | 0.8 s | 0.4 s | 3.3 s | 1.6 s |
| 4 | 524288 | 14.1 s | 7.1 s | 4.6 s | 2.3 s |
| 5 | 128 x 4092 | 14.8 s | 7.5 s | 32.9 s | 17.3 s |
| 6 | 262144 | 115.3 s | 62.4 s | 96.0 s | 51.4 s |
| 7 | 10 x 32768 | 9.1 s | 4.6 s | 40.9 s | 23.8 s |
| 8 | 131072 x 4 | 13.4 s | 7.2 s | 11.1 s | 5.6 s |
| 9 | 512 x 512 | 3.0 s | 1.6 s | 2.7 s | 1.4 s |
| 11 | 262144 | 19.4 s | 9.9 s | 8.9 s | 4.4 s |
| 12 | 1023 x 511 | 2.0 s | 1.3 s | 12.7 s | 7.3 s |
| 13 | 262144 | 35.2 s | 17.5 s | 8.9 s | 4.5 s |
| 14 | 262144 | 1.8 s | 1.0 s | 13.6 s | 6.9 s |
| 15 | 16384 | 6.2 s | 3.2 s | 3.5 s | 2.5 s |

The SIMD architecture takes advantage of the very fast realization of broadcasts and reductions. In case of the CM the performance of many local operations is also slightly better than on the iPSC/860. This is especially true for the intrinsic functions sin, cos, sqrt and exp.

But for many problems MIMD gives better performance due to the asynchronous execution of local computations and local communications. Processes will only be synchronized with broadcasts and reductions and not with every instruction.

# 6 Optimization Issues

Though a dependence analysis has been realized to verify that sequential loops are equivalent to the parallel loops, the current version of Adaptor has no optimizations implemented. Some first experiments have shown that the following optimizations issues are the most important ones:

- optimization of the high level communication operations of the DALIB,
- loop fusion to make register optimization possible within the compiler,
- overlap analysis to reduce memory transfer,
- minimizing the use of temporary arrays,
- combination of messages to avoid communication overhead,
- and overlapping computation and communication.

This kind of optimization techniques will be investigated in the next future for more complex applications.

# 7 Summary

Adaptor is a prototype of a compilation system for High Performance Fortran that gave the following insights:

- It is possible to translate efficient CM Fortran or SIMD programs to efficient message passage programs for MIMD architectures. This guarantees portability and efficiency of all existing data parallel programs.
- The potential of parallelism is mainly given by array operations and the parallel loops. Automatic parallelization has less importance, program rewriting of sequential programs will be necessary in any case.
- Experiments on shared memory and virtual shared memory systems like the Alliant FX/2800 and the KSR 1 have shown that processes with an own address space are more efficient than programs based on threads with a common address space. As the compiler knows that no data is shared, data accesses can be more optimized by using registers and local caches.

Adaptor itself can and will be used to implement different optimization strategies for High Performance Fortran compilers. Most interesting are optimizations that depend on system parameters like communication start up time, communication bandwidth and so on. Optimizations at run time are also of great interest.

Adaptor will also be used to define and test language extensions of interest that could be part in one of the next versions of High Performance Fortran.

**Acknowledgements**

I thank the Central Institute for Applied Mathematics at the research center in Jülich for providing the iPSC/860 and Renate Knecht for her user support.

The following people have influenced this work by valuable discussions: Clemens-August Thole (GMD), Rolf Hänisch (GMD), Michael Gerndt (ZAM, Research Center Jülich), John Merlin (University of Southampton), Dave Watson (NA Software, Liverpool) and James Cownie (Meiko, Bristol).

Many thanks are also due to Falk Zimmermann for his implementation work and for the discussions about proving correctness of the transformations realized within Adaptor.

# References

[Bra92] T. Brandes. ADAPTOR Language Reference Manual. Internal Report ADAPTOR-3, GMD, 1992.

[BS87] T. Brandes and M. Sommer. Realization of a Knowledge-Based Parallelization Tool in a Programming Environment. In *International Conference on Supercomputing*, Athens, Greece, June 1987.

[CCRS91] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A Parallel Programming Environment for Scientific Applications Using Communications Structures. In *Proc. of 1991 International Conference on Parallel Processing*, St. Charles, Illinois, August 1991.

[Cor90] Thinking Machines Corporation. Connection Machine Model CM-2. Technical Summary Version 6.0, TMC, November 1990.

[Cor91] Thinking Machines Corporation. CM Fortran Programming Guide, Version 1.0. Manual, TMC, January 1991.

[FHK+91] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90079, Department of Computer Science, Rice University, April 1991.

[Fox91] G. Fox. Achievements and prospects for parallel computing. *Concurrency: Practice and Experience*, 3(6):725–739, December 1991.

[GE90] J. Grosch and H. Emmelmann. A Tool Box for Compiler Construction. *Lecture Notes of Computer Science*, 477:106–116, October 1990.

[Ger89] H.M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, 1989.

[Gro91] J. Grosch. Puma - A Generator for the Transformation of Attributed Trees. Compiler Generation Report 26, GMD, Forschungsstelle an der Universität Karlsruhe, 1991.

[HE91] M. Heath and J. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[HKT91] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. Technical report, Department of Computer Science, Rice University, 1991.

[HLJ+91] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and R. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 1991.

[KLS91] K. Knobe, J. Lukas, and G. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1991.

[Koe92] C. et al. Koelbel. Draft of High Performance Fortran Language Specification. Technical Report Version 0.4, Department of Computer Science, Rice University, October 1992.

[Mer91] J. Merlin. ADAPTing Fortran 90 Array Programs for Distributed Memory Architectures. In *Proc. 1st International Conference of the Austrian Center for Parallel Computation*, Salzburg, October 1991.

[MFL+92] A. Mohamed, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Lu, N. Lin, and N. Yeh. Applications Benchmark Set for Fortran-D and High Performance Fortran. Technical Report 327, Northeast Parallel Architectures Center, 1992.

[O'R90] T. O'Reilly. *X Toolkit Intrinsics Reference Manual.* Nutshell Handbooks, Sebastopol, CA, 1990.

[Sun90] V. Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 3(10), December 1990.

[WF91] M. Wu and G. Fox. Compiling Fortran 90 programs for distributed memory MIMD parallel computers. Technical Report No. SCCS-88, Syracuse Center for Computational Science, 1991.