Linguistic Support for Heterogeneous Parallel Processing: A Survey and an Approach

Charles C. Weems
Glen Weaver
Steven Dropsho

Computer Science Department
University of Massachusetts
Amherst, MA 01003

Author's E-mail Addresses
Weems@cs.umass.edu
Weaver@cs.umass.edu
Dropsho@cs.umass.edu

Abstract

Coding a highly parallel application to run on a heterogeneous suite of processors (both metacomputers and mixed-mode computers) with high efficiency, ease of implementation, and portability is a significant challenge. This paper first surveys recently proposed and existing parallel languages from the perspective of programming complex, heterogeneous systems. We then propose two essential features to be included in programming languages that are intended to support heterogeneity.

1. Introduction

Recent examples have shown the success of combining heterogeneous computing hardware to solve complex problems[KMCP93][NGH+91]. When the architecture of the machine matches the structure of the problem, the algorithmic solution is often easier to develop and can execute more efficiently. The currently popular practice of networking existing machines is merely a beginning [SC92]. As hardware continues to diminish in size and cost, new possibilities are being created for systems that are heterogeneous by design. Examples of mixed-mode heterogeneous systems are beginning to appear. The CM-5 and the Image Understanding Architecture (IUA) [Wee89] are two examples of architectures that incorporate different modes of parallelism within a single machine. In the future, we can expect even more complex examples of heterogeneity.

Driving the growth of these complex combinations of computer hardware are complex applications: vehicle navigation, air traffic control, intelligent information retrieval, environmental modeling, turbulent fluid flow modeling, etc. Even if the hardware is built, the application program must be written, debugged, and maintained by humans. A programming facility is needed that simplifies the development task while maximizing performance [KPSW93]. The programmer must be able to specify the algorithms at a sufficiently abstract level so that he or she is not overwhelmed with trivialities yet has access to those details that are needed by system software (compiler, linker, and run-time support) to efficiently execute the code. Furthermore, as applications become more sophisticated, debugging becomes more complex. Hence, the program developer needs considerable support from the debugger to sift through the plethora of information.

The programming language is the interface between the programmer and the machine. The power and expressiveness of the language affect the efficient use of the programmer's time and the machine. In any environment a single programming language is desirable for reasons of programmer efficiency. However, in a heterogeneous programming environment, this one language must be able to express many useful models of computation since a single model is insufficient for expressing all forms of computation efficiently for all target hardware architectures. Examples of models to be supported include: sequential, control parallelism, coarse and fine grained data parallelism, shared memory, distributed memory, dataflow, etc., and orthogonal combinations of models.

A single application can include many forms of concurrency each requiring a different programming model in order to be expressed in a usable manner[Wee93b]. These differing models must both be expressible in the same program and in a hierarchical encapsulation of individual models. With this capability the programmer has only to work with the appropriate model of abstraction at the appropriate level (e.g. control parallel model at a high level, but a data parallel model at a lower level within a control thread), thus simplifying the programming.

The level of parallelism of large applications makes the tasks of communication, synchronization, mapping, scheduling, and load balancing significant chores for the programmer. These are tedious and error prone tasks that are ideal for the dogmatic compiler and runtime system. However, automatic discovery of algorithmic properties from the code is non-trivial, so mechanisms should be available for the programmer to give semantic information to these tools that assists them in finding the most efficient implementation.

Fixed programming languages suffer from limitations on their adaptability to unanticipated machines, algorithmic models and data structures. In many cases these require new semantics be added to the language for efficient implementations to be automatically generated. Ideally, a programmer will not only be able to build new data types but also add semantics that the compiler and system can use for optimization and mapping information. Thus, a language must be extensible beyond the ability to combine existing primitives as current languages provide.

One key to ease of programming is that code is reusable and, therefore, portable. To port between radically different machine models poses a significant problem. The solution is to provide translation and emulation capabilities across computational models to permit execution of one machine model on another machine, albeit possibly with low efficiency. Thus, the unmodified code can be run immediately and then be fine tuned (or rewritten) later if the performance is unacceptable.

In the last section of this paper, we further address the features that we believe are necessary for a heterogeneous parallel programming language that bridges the gulf between the programmer and a complex heterogeneous machine. The next section is a survey of prominent parallel languages in which we consider how each one addresses (or fails to address) the issues described above.

It should be noted that this paper addresses the linguistic features required for a language to capture the programmer's knowledge of an applications to the level necessary to automatically provide an efficient implementation on a heterogeneous system. We do not discuss HAsC, DHSS, or HBD, which address mechanisms for using this information in a heterogeneous environment at run time. Nor are we proposing specific language extensions to support these features. The implementation of the features is an open research issue and our purpose here is merely to motivate their necessity.

2. Language Survey

2.1. Cluster-M

Cluster-M [ES93] [EF93] is a program partitioning and placement model for heterogeneous processing, not a language. It uses the tree structure to unify the descriptions of programs and physical systems. A program is represented as a tree structure, called a Cluster-M specification, where each leaf node is a single computation operand. Nodes are called clusters and related nodes are grouped to form a cluster at the next higher level. All clusters at a level are independent and may be executed simultaneously. This hierarchy of clusters enables fine to coarse grain data parallelism. The synchronization of parallel events is controlled by the system. The definition and use of data types is not discussed.

The manner in which a program is clustered is specified by the programmer and may be modified during runtime. This gives the user the ability to express how clusters (hence data) are related but there is no mechanism for explicitly expressing how to distribute clusters among the processors; the mapping is to be automatic. Even though Cluster-M represents the physical system as a tree structure in a similar fashion as the program, automatic mapping of clusters is an unsolved, non-trivial problem.

2.2. High Performance FORTRAN

High Performance FORTRAN (HPF) is a set of extensions to FORTRAN 90 [Lov93] for fine to medium grain data parallelism. HPF inherits synchronous data parallelism from FORTRAN 90 but adds support for data distribution, communication, and the *FORALL* statement.

The programmer assists in data distribution by specifying a view of the processor array as an array of virtual processors, the alignment of arrays with respect to one another, and a distribution of array elements to virtual processors. This distribution mechanism is restricted to rectilinear arrays. For communication between virtual processors, HPF provides library routines for reduction, scatter, parallel prefix, parallel suffix, and sorting, as well as, implicitly handling accesses to array elements on other processors. The programmer expresses parallelism with the *FORALL* construct, which performs implicit synchronization. HPF does not allow for data type abstraction.

2.3. Delirium

Delirium [LS90] is a coordination language that embeds sequential sub-computations written in C or FORTRAN. It is a more restrictive coordination language than Linda or PVM but it has the benefit of deterministic execution and reuse of existing code. The language is designed for medium to coarse grain control parallelism and has a shared memory abstraction. The user must state all variables each routine will destructively modify so the system can do dynamic dependency analysis and automatic scheduling of parallel processes. Some control of process mapping is possible.

2.4. Linda

Linda [Gel85] defines a shared, associatively addressed, semi-persistent store called tuple space that holds tuples. Tuples can contain anything and the system transparently manages dispersing the tuples. Tuple space effectively replaces message passing and shared memory. A Linda program is initiated by putting tuples containing code in tuple space. These tuples are then extracted from tuple space and executed. Thus, Linda is inherently control parallel, but data parallelism is also easily expressed. Because Linda's abstraction does not reveal the time for tuple space access, Linda supports all levels of granularity but is more efficient with a coarse grain size.

Linda provides only basic facilities for communication without providing for higher level abstractions of communication patterns. Similarly, synchronization abstractions (e.g. locks) must be built from the atomicity of tuple access. Linda does not address data typing issues.

2.5. PCN

PCN [FOT93] is a parallel programming language based on C and FORTRAN but adds the extensions of parallel composition and definitional variables. Parallel composition is the use of hierarchical abstractions by the programmer to define an arbitrary parallel model. These virtual topologies have a parallel abstraction at the top level and relegate the physical mapping to the lowest level, enabling the user to define the mapping to processors. Both control and data parallelism are supported in the language.

Communication and synchronization are provided by definitional variables that provide a restricted shared memory view to the programmer. A definitional variable sets up a communication channel between processes and the synchronization is data-driven. The communication has low overhead and each process is lightweight (tens of instructions to create or migrate), so relatively fine grain parallelism is possible. Communication, synchronization, and scheduling details are implemented by the compiler.

The compiler allows users to implement application-specific extensions to the PCN language.

2.6. PVM

PVM [Sun90] is not a language so much as a run-time facility that provides a standardized set of control parallel programming extensions to sequential languages. PVM provides an abstraction of a network of computers as a single MIMD *virtual computer* suited for coarse-grain parallelism. The programmer uses a set of library routines that provide message passing, selective broadcasting, locks, barriers, and process control. PVM also handles conversion of base type (e.g. integer and floating point) representations (e.g. big-endian to little-endian)

When a new process is spawned, its executable file is designated by a logical name. The programmer supplies a table which has a many-to-many mapping of logical names to specific file names. This allows PVM to find alternate versions of a given portion of code (e.g. versions compiled for different machines). PVM also allows a spawned process to be placed on a particular machine or class of machines. Otherwise, PVM uses dynamic load-balancing to distribute the process, but once assigned processes do not move.

PVM's abstractions for communication, synchronization, and distribution are low level; thus, the programmer must build higher level abstractions. PVM does not address data type issues.

2.7. p4

P4 [BL93] is a set of macros that provide functionality similar to PVM. P4 also presents a network of computers as a single "virtual" computer. However, p4 is more concerned with efficiency than PVM. The p4 user views the virtual computer in one of three ways: as a distributed memory machine, a shared memory machine, or as a cluster machine.

P4 provides message passing for distributed memory machines. For shared memory machines, p4 provides a monitor data type that is used instead of locks to control access to the shared memory. In the interest of efficiency, p4 does not provide monitors unless real shared memory exists.

Cluster machines are distributed machines in which the nodes themselves may be shared memory machines (e.g. a network of SGI multiprocessors). Monitors may be used within a cluster, but only message passing may be used between clusters.

Like PVM, p4 is intended for coarse grain, control parallel applications and does not address type definition issues. P4 does add shared memory as a communication abstraction and monitors as a synchronization abstraction to access shared memory, but does not greatly improve the level of abstraction. In p4, users have even less flexibility in placing processes on processors than in PVM.

2.8. C**

C** [LRV92], based on C++, is an object oriented data parallel language for fine to coarse grain parallelism. C** offers the SIMD computational model benefit of nearly race-free execution without its restrictions of lockstep processing. The aggregate class declares an ordered collection of values which may be operated upon concurrently. Aggregate structures are restricted to multidimensional arrays. C** data abstraction features are those of C++ with extensions for aggregates.

The user may slice aggregates along the dimensions thus controlling the grouping of values to be automatically distributed by the system. The code for communication and synchronization to implement parallel functions is automatically added by the compiler.

2.9. PC++

PC++ [LG91] is an object oriented programming language for expressing medium to coarse grain data parallelism. The model is the Distributed Collection Model that represents a data structure as a collection of homogeneous elements and allows the programmer to specify how the data structure is to be partitioned and distributed among the processors. In the current implementation this feature is restricted to rectilinear structures. Communication is through messages and synchronization is implemented through the compiler.

PC++ lets the element class inherit knowledge of the algebraic and geometric structure of the collection. The programmer can use such knowledge to develop routines (such as parallel prefix) for an element class based on the existence of properties of the element class (such as associativity) without having to know the implementation details of the element class. There are no rules for structuring such knowledge and there are no mechanisms by which the system might use this knowledge.

2.10. ABCL/1

ABCL/1 [YBS86] is an object oriented parallel programming language based on the Actor model for coarse grain control parallelism. In the ABCL/1 each object has is considered to have its own processing power and persistent memory. Communication is via messages and the objects retrieve messages from queues. The script construct defines the valid messages an object will accept and the actions associated with each valid message type.

Although the message abstraction is explicit to the user, its implementation details are hidden. Synchronization is performed through messages.

There are no explicit mechanisms for influencing the placement of processes nor enhancements to standard type definition methods.

2.11. Jade

Jade [RSL93] is an object programming language that is designed for exploiting coarse grain control parallelism in a shared memory system. Parallelism is implicitly stated in a serial program and is dynamically extracted. The programmer, however, must supply information about the data accesses of each task. The *withonly* construct lets the programmer define a task, the shared variables accessed by the task, and the type of access (read, write, or read-write) to each variable. During runtime this information is used in a dependency analysis to determine the available parallelism and to schedule tasks for deadlock free execution. The *resource* construct allows the programmer to dictate the host on which a task should be run for very coarse process assignment control.

Based on C, Jade does not extend the type definition capabilities beyond those in C.

2.12. Ada 9X

Ada 9X [Dob93] augments Ada 83's tightly coupled, medium grain task model with constructs for coarse grain control parallelism and operator overloading for fine grain data parallelism. Used in combination, these features give the programmer a hierarchy of parallelism models. Control parallelism in Ada 9X closely follows a distributed computing model. Ada 9X defines *active partitions* that are equivalent to an Ada 83 main program. Active partitions communicate and synchronize via Remote Procedure Calls (RPC). Ada 9X also provides *passive partitions* that contain shared data and code. However, Ada 9X does specify how the programmer can control the distribution of passive or active partitions.

Ada provides data abstraction and overloading of operators, but this does not provide the semantic knowledge needed by the compiler for optimization.

2.13. Charm++

Charm++ [KK93], which is based on C++, encompasses both control and data parallelism. Charm++ attempts to divide up the work of managing parallelism between the programmer and the run-time system. The programmer is responsible for specifying the parallelism, but the run-time system takes care of distributing the work load (via dynamic load-balancing) and establishing communication links.

Objects in Charm++ communicate using messages. To hide latency, Charm++ uses *split-phase* communication in which the sending object does not expect a response. Rather, any response is handled as a separate message. This supports control parallelism. However, Charm++ also defines a special object which has replicated instances on each processor and coordinates among the instances to provide for data parallelism. Thus, Charm++ supports medium to coarse granularity and provides implicit synchronization.

Charm++ prohibits global variables, but provides for a limited set of global communication primitives. The user is provided with read-only, write-once, accumulator, monotonic, and distributed table data objects. These can be refined via inheritance to work with any type. C++ gives Charm++ the usual and desirable data abstraction capabilities of an object-oriented system, but does not provide for any additional semantic information. Charm++ permits only crude data distribution control and no control over process distribution.

2.14. Mentat

Mentat [Gri92] [Gri93], based on C++, is an object oriented language for defining medium to coarse grain control and data parallelism. The model supports only messages between objects, but the compiler automatically generates the code for all communication and synchronization. Following a coarse-grained dataflow model, member function calls to objects are nonblocking unless data dependencies cause a conflict.

Mentat uses a virtual machine model that facilitates porting of applications. However, the Mentat model of medium to coarse grain parallelism restricts the families of machines on which Mentat code will run efficiently.

In data parallel objects, Mentat introduces programmer-written call-backs that are member functions which dictate how any user defined data structure should be partitioned. The system uses the call-back functions along with system parameters to dynamically partition a data structure for parallel processing with automatic load balancing. Through the call-back definition the user explicitly controls the distribution of data. Since Mentat is based on C++, the data abstraction capabilities are the same as C++ with the novel addition of data partitioning semantics added via call-backs.

2.15. Summary of Survey

A summary of the survey is shown in Table 1. A fictitious ideal language is included in the last row. This table suggests where a language either eases programming or runs efficiently. Any language that expresses control parallelism can express data parallelism, but those languages that also provide a separate abstraction for data parallelism are designated as *data-and-control*. Similarly, the grain size column suggests which grain size the language's abstraction can support efficiently. A language's communication is implicit when the programmer does not have to program the communication or even know it is occurring. Languages that synchronize implicitly insert synchronization points automatically. Two important properties of data type extensibility are that new types can be created without resorting to base types and that the compiler can optimize instances of the new type.

The first three columns show the wide diversity between the target architectures of the programming languages. Few languages offer both data and control parallelism. Those that do either restrict the granularity of parallelism or offer only one abstraction model for communication. An ideal heterogeneous parallel language needs all of these options to be able to represent every existing computational model efficiently. Thus, it must offer all types of parallelism at each appropriate level of granularity as well as multiple communication abstractions to allow the user to program in the most efficient model for each portion of the application.

Some languages enable the user to specify process or data mapping to processors, none do both. Certainly, shielding the user from distribution details eases the effort of development but supplying mechanisms for the user to guide the mapping process is essential for exploiting the user's knowledge of the application. Combining the system's detailed analysis with the user's higher level knowledge should result in more efficient mappings than either one alone is capable of achieving.

Synchronization is performed many different ways by the languages, but not all methods are expressible in any single language. Giving the user mechanisms for synchronization and the ability to create tailored synchronization abstractions for an application again allows the user to impart knowledge about the application and its behavior. Automatic control of synchronization eases the burden on the programmer but one abstraction cannot satisfy the requirements of all architectures for efficient synchronization. Ideally a language should be extensible in this respect allowing for novel ideas to be added.

Language	**Preferred Model of Parallelism**	**Grain Size**	**Abstraction of Communication**	**Abstraction of Process/Data Distribution**	**Abstraction of Synchronization**	**Extensibility of Data Types**
Cluster-M	Data	All	Clusters in a tree	No	Implicit	NA
HPFortran	Data	Fine to medium	Library routines for global operations and implicit by access	Data distribution for arrays	Yes, implicit in FORALL	Fixed
Delirium	Control	Medium to coarse	Shared memory	No	Implicit	No
Linda	Control	All	Shared tuple space but no patterns	No	Tuples are accessed atomically	NA
PCN	Data and control	Fine to coarse	Definitional variables	Yes	Implicit	No???
PVM	Control	Coarse	Message passing with multicast and broadcast	Can force process assignment to specific nodes	Locks and barriers	NA
p4	Control	Coarse	Message passing or shared memory with a hybrid available as well	No	Locks, barriers, and monitors	NA
C**	Data	Fine to coarse	Implicit	Partitioning of arrays	Implicit	Yes, limited to combinations of fixed types
PC++	Data	Medium to coarse	Implicit	Data distribution for arrays	Implicit	Yes, limited to combinations of fixed types
ABCL	Control	Coarse	Message passing among active objects	No	Via messages	Yes, limited to combinations of fixed types
Jade	Control	Coarse	Shared memory	Slight	Implicit	No
Ada 9X	Data and control	All	RPC for coarse grain but no patterns	No	Based on RPC	Yes, limited to combinations of fixed types
Charm++	Data and control	Medium to coarse	Message passing with fixed global operations	No	Via messages	Yes, limited to combinations of fixed types
Mentat	Data and control	Medium to coarse	Message passing	Yes, via call-backs	Yes, data driven and hidden from user	Yes, limited to combinations of fixed types
ideal	Any	All	Allows user to specify details at one level, yet hidden at others	Yes, user can specify patterns to aid distribution	Yes, user can define and then hide as abstraction	Able to define new first class types

Table 1: Summary Comparison Information

PCN allows the addition of application-specific extensions to the language into the compiler. None of the other languages surveyed allow the definition of new data types and associated semantics usable by the system for optimization and mapping. Many provide object oriented data abstraction capabilities. While the encapsulation of data is very useful for the user the information is not in a form that the system can exploit. For example, it has been demonstrated in many languages that revealing the semantics of multidimensional arrays to a system (by rigidly embedding semantics in the compile) can be of significant help to the user in making code more efficient.

3. An Approach to Linguistic Support for Heterogeneous Processing

Each of the languages described in the preceding section has certain limitations in the context of heterogeneous processing. One limitation we have noted is a lack of support for more than few models of parallelism. Another is a lack of support for more than one target architecture, both in terms of code generation and distribution of code and data. We examine these limitations further in this section, and propose language features that must be present to fully address them. It is important to again note that we are not specifying a language in this paper, but merely the capabilities that we believe will be necessary in whatever language is eventually developed to support general heterogeneous parallel programming.

3.1 Support for Multiple Models of Parallelism

Many of the languages are limited to or strongly favor a particular model of parallelism. Some support multiple models, but either as a fixed set in the language (e.g. Ada 9X, Charm++, Mentat), combinations of predefined models (PCN), or through reliance on external languages (e.g. PVM, P4). Most of the languages that support data and control parallelism are based on a common (e.g. MIMD/SPMD) model, and do not support synchronous data parallelism (i.e. SIMD) directly; HPF being a notable exception.

Obviously, in a language intended to support truly heterogeneous processing, there should not be such limitations on the use of different models of parallelism. Beyond the mere appeal to linguistic and algorithmic aesthetics, there is a fundamental reason that a heterogeneous parallel programming language must not restrict its support to certain models of parallelism: in order to produce efficient object code, a compiler must be able to extract a certain amount of semantic information from the source code. If the necessary semantics cannot be expressed in the language, or are obscured in some way, then the compiler is unable to apply appropriate optimizations.

As an example of obscured semantics, consider Ada 9X, which provides two different mechanisms to support control parallelism. In addition, the Ada 9X extensions provide object-oriented programming capabilities. Thus, a programmer can define a new type, such as a matrix, that appears to be a SIMD parallel construct. For example, the arithmetic operators could be overloaded to perform matrix addition and multiplication directly in Ada expressions. The implementation of the matrix package could even call external run-time libraries that execute on parallel hardware. *But the Ada compiler is unable to perform any optimization of the matrix code.* It has no way of knowing that common subexpression elimination can be employed on matrix expressions because the semantics of the operators have been obscured -- they are merely function calls. It cannot schedule the registers in the parallel machine because it has no information about the target hardware. Placement and partitioning of the data cannot be considered at compile time for the same reason.

As a second example, consider that the tuple space in Linda can be used to construct synchronization mechanisms such as locks or semaphores. However, because the semantics of these constructs are lost in their implementation, the system has no way of optimizing their execution through compile-time methods. In a simple communication example, such as shifting an entire array by one position in a given direction, the semantics of the operation are lost in a series of tagged input and output operations to tuple space, so that a compiler has no way of knowing that the operation is highly parallel if properly orchestrated. Even if Linda is viewed in the context of being a coordination language that depends on a separate computation language to support such orchestrated computations, if the programmer wishes to use Linda to further distribute the array computations, there is no way for the computation language to communicate information to the coordination language that enables it to automatically optimize its coordination. Instead, the programmer must build more sophisticated coordination on top of Linda's primitives, which suffers from the problem noted above.

Even if a language has an enriched set of built-in models of parallelism, it may not be able to efficiently support new models that are derived from developing applications. Whenever a language distinguishes between a set of first class models whose semantics are known to the compiler and second class models that are built on top of the first class set with no additional explicit semantic information, then optimization of the second class models is difficult if not impossible in some cases. If we consider the amount of effort that has gone into parallelizing compilers for FORTRAN loops as merely the optimization of translating one programming model into another, and multiply that effort by the cross product of all of the models in a heterogeneous system, the difficulty becomes more apparent.

Programmers are continually developing new ways to exploit parallelism in problems, and the language must provide mechanisms for those models to be expressed so that they can be efficiently translated. In order to reduce the effort of compilation, it should be possible for the programmer to define new models with sufficient semantic information that they are effectively adding new first-class constructs to the language. This capability goes beyond the normal concept of extensibility in that the result is actually a change in the form of the language. Hence we refer to it as *metamorphism.*

3.2 Metamorphism

A language that supports metamorphism enables the user to define new first-class models of parallelism. For example, a user might define a new data-parallel model, such as a triangular matrix, by creating a new first-class type with a set of parallel operations.

The language supports the creation of a new first-class type by enabling the user to associate with the type parameterized identifiers from a standard set of semantic tags. All that the tags do is inform the compiler of certain properties that are necessary to trigger particular optimizations. For example, a Square_Geometry tag might trigger particular data partitioning and placement optimizations.

In addition, the user should be able to define new tags and optimizations. For example, they might provide a transformation that converts a pair of Triangular_Geometry-tagged objects into a Square_Geometry object so that the existing partitioning and placement optimizations can be applied. Alternatively, the programmer might define a new partitioning and placement optimization that operated directly on Triangular_Geometry objects. Such transformations are, of course, more likely to be predicated on matching multiple required semantic properties to tags and attributes associated with a compilation object.

The transformation's output is also a compilation object with a set of tags and attributes, indicating properties that it preserved or added. Furthermore, for debugging purposes, auxiliary data required to perform a reverse transformation should be output for later use.

A compiler for a metamorphic language thus takes the form of a framework into which the user can insert additional transformations. The base language merely consists of a predefined set of constructs and their associated transformations. The job of the compiler then becomes more of a planning function: given a source module and a particular target in the system, it applies means-ends analysis to establish the optimal series of transformations to produce an object module. A display of the transformation chain, possibly annotated by the analyzer, may even be a useful tool for the programmer to identify further transformations that may be needed to improve optimization. Such a tool would probably be augmented with a browser to enable the programmer to examine existing transformations for their applicability and thus facilitate reuse.

Among the transformations that must be supported to enable portability are those that retarget code from one model of parallelism to another. Some of these transformations are static, such as embedding a mesh into a hypercube. Others require dynamic transformation such as emulation of a SIMD array on a MIMD processor. As noted in a previous section, with sufficient emulations, it is possible to run any algorithm on any machine with the expected loss in efficiency. Thus, code can be ported to a new configuration without change, where its performance can then be analyzed and augmented. How the augmentation is achieved raises the second limitation of many of the surveyed languages: support for multiple targets.

3.3 Support for Multiple Target Architectures

Most of the languages we surveyed assume a single, homogeneous parallel target architecture. The exceptions are found among the coordination-oriented systems [GC92] such as PVM, P4, Linda, and Cluster-M. But although these enable a heterogeneous ensemble of architectures to work together, they do little to facilitate the portability and retargetting of code between architectures. It is assumed that the computation is programmed in other languages, and facilities are provided to allow user-programmed partitions to communicate homogeneously (in the case of PVM, P4, and Linda) or to automatically manage the partitioning and placement of user codes onto appropriate targets (using Cluster-M).

It is understandable that automatic retargetting of code has not been addressed when we consider that even a very modest change in the architecture of a target can result in a completely different algorithm. Indeed, even for the same target different algorithms can be required to achieve optimal performance depending on the data set. For example, one sorting algorithm may be optimal for large data sets and another for small sets; a choice of search algorithm may depend on whether the data is partially sorted, etc. Thus, beyond static transformations and emulations, support for multiple targets becomes a matter of managing multiple algorithmic implementations of the same function.

Mechanisms for dealing with heterogeneous collections of algorithms for a single function have long been employed in database systems. The key to these mechanisms has three parts: (1) the algorithms must share a common logical interface, (2) the algorithms must be identified as alternate implementations of the same function, and (3) a means of choosing between them at run time must be provided.

We can thus use the same sort of mechanism to manage multiple algorithmic implementations for heterogeneous targets. However, there are two additional requirements to facilitate compilation: (1) The interface must supply sufficient semantic information to enable external transformations to be applied to the implementations as if they are identical, and (2) the alternate implementations must carry additional semantic information specific to each so that transformations can crack the abstraction shell and determine the essential differences between the implementations even if they are in a precompiled form.