A Methodology for Programming Scalable Architectures *

Rajendra Panwar and Gul Agha †
Open Systems Laboratory
Department of Computer Science
1304 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {panwar | agha }@cs.uiuc.edu

*The research described has been made possible by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

†The authors thank Daniel Sturman for his development of Broadway and Svend Frølund, Daniel Sturman, Wooyoung Kim, Shangping Ren and other members of the Open Systems Laboratory have provided helpful suggestions and discussions. The authors would also like to thank the referees for providing several constructive suggestions.

Abstract

In scalable concurrent architectures, the performance of a parallel algorithm depends on the resource management policies used. Such policies determine, for example, how data is partitioned and distributed and how processes are scheduled. In particular, the performance of a parallel algorithm obtained by using a particular policy can be affected by increasing the size of the architecture or the input. In order to support scalability, we are developing a methodology for modular specification of partitioning and distribution strategies (PDS's). As a consequence, a PDS may be changed without modifying the code specifying the logic of a parallel algorithm. We illustrate our methodology for parallel algorithms that use dynamic data structures.

1 Introduction

In parallel computing, the set of operations and the partial order in which they may be carried out define an *ideal algorithm* [25]. We use Actors, a form of concurrent objects, to specify ideal algorithms. Actors naturally express an ideal algorithm without introducing unnecessary sequentiality in the code. Data parallelism can be naturally expressed by messages broadcast to a group of actors. Functional parallelism may be expressed by sending messages concurrently to a number of actors and using a join continuation actor to synchronize their responses [2]. Moreover, because actors are message-driven, they naturally overlap communication and computation, thus masking latency whenever feasible. This provides a form of pipeline concurrency [4].

Because of limitations on computation and communication resources in practical architectures, implementations of a parallel algorithm may not use all the parallelism available in the ideal version of the algorithm. A scalable model of parallel computing is realized in multicomputers. Each node in a multicomputer may be sequential and may carry out many actions which can be potentially executed in parallel. In particular, how the data is placed determines which potentially parallel operations in an algorithm may be executed sequentially. Assuming a computational model which measures performance while taking into account particular network characteristics, different placement policies lead to different performance.

In this paper, we describe a methodology for programming concurrent computers which allows separate specification of an ideal algorithm and the policies used to place (or migrate) the computational objects that have been created to realize the algorithm. The two

specifications, namely the ideal algorithm and the placement (or migration) policy, may be combined to obtain an efficient implementation of the algorithm for a given problem size and architecture (see figure 1 [7]).

We call the policies used to place and migrate actors representing a data structure a partitioning and distribution strategy (PDS). Our methodology allows expression of PDS's for static as well as dynamic data structures. The PDS's are expressed as first class objects which can interact with the objects representing the ongoing computation and decide where to place newly created objects and how to migrate existing objects to optimize resource utilization.

Thus the methodology allows an application programmer to develop software for an ideal algorithm without specifying the details of architecture dependent PDS. Because the code specifying an ideal algorithm is architecture independent, it may be reused when ported to a different architecture. Similarly, code for a PDS may be reused with a number of ideal algorithms which use a group of actors representing the same abstract data type.

The organization of this paper is as follows. In the next section we motivate our work. Section 3 describes our computational model and our methodology. Section 4 illustrates our methodology with a detailed example. Section 5 presents the state of implementation and gives performance results for some benchmarks. Section 6 relates our work to the previous work and discusses future work.

2 Partitioning and Distribution Strategies

In this section we describe the significance of PDS's and the need for modularity in the specification of PDS's. Specifically, we argue that:

- The use of a specific PDS for an ideal algorithm can significantly affect performance.
- The efficiency obtained by using a PDS is related to the characteristics of an architecture and algorithm. In particular, scaling up an architecture or the input size for an algorithm can change the efficiency obtained by using a given PDS.
- In some cases, increased efficiency may be obtained through the use of *dynamic* PDS's which require actors to be migrated during a computation.

Several examples in the literature illustrate the importance of architecture specific PDS's for obtaining high performance on scalable architectures. Consider finite difference algorithms that solve Partial Differential Equations using a data structure such as a two-dimensional array representing the values at various grid points of the input domain. In these algorithms, the value at a grid point is iteratively updated using only its old value and the values of a small number of neighboring grid points. In mapping the input array to the nodes of a multicomputer, overhead can be significantly reduced if neighboring grid

points are mapped to the same processor or to neighboring processors. For example, an implementation of the 3D Navier-Stokes equation based on the explicit MacCormack scheme yields 90% efficiency on a 1024 processor NCUBE using a Gray code mapping for ensuring nearest neighbor communication. Without the use of Gray code mapping, the efficiency falls to 68% [37].

The choice of a PDS is affected by scalability in two ways. First, the best PDS may be a function of the architecture and problem sizes. For some parallel algorithms, when the architecture size is scaled up, different PDS's may become more efficient. For example, in dense Cholesky factorization, a load balanced PDS is more efficient when the load is sufficiently high. If the architecture is scaled up significantly, the communication overhead becomes more significant. A PDS providing greater locality is more efficient in that case [5].

Second, the best PDS may be a function of the input size because the input size may influence the choice of the target architecture. Consider again the dense Cholesky decomposition of a matrix. Figure 2 shows an analytic estimate of the time taken as a function of the number of processors for both a network of workstations (with slow broadcasting), and a mesh multicomputer. The PDS used in calculating this estimate is cyclic-checkerboard and results are based on an analytical model as described in [29]. The analysis suggests that in order to reduce the total time taken to 1000 seconds for a $20,000 \times 20,000$ matrix, we may use a mesh multicomputer with a fast network but not a network of workstations. In this case, as the input size is increased, a different architecture and hence a different PDS would be suitable.

For certain applications, an optimal mapping at one stage of the algorithm may be non-optimal for another. In such cases, objects need to be migrated during execution in order to improve performance. For example, using 32 processors to solve Navier-Stokes equations using the Beam-Warming algorithm, an implicit time-marching scheme, the execution efficiency is 28% with fixed mapping [9] and 80% using migration [33]. As we will describe later, migration is required by a PDS which is dynamic.

This discussion illustrates the importance of using an appropriate PDS to obtain efficient utilization of resources for many algorithms. Two other points may be emphasized. First, because the efficiency of a PDS is affected by scaling an architecture or problem size, modular specification of PDS is not only good software engineering but supports increased portability of code. Second, a PDS may involve complex interactions with the on-going computation, for example, in triggering migration based on phases of an algorithm or global load conditions. We therefore represent a PDS by a collection of actors which may observe and delay messages to actors, and place or move them.

3 Computational Framework

As noted earlier, we use Actors as our computational model of concurrency and extend it with constructs to specify placement. Actors are message driven objects which unify data and the operations modifying the data. Actors are self-contained, interactive components of

a computing system that communicate by asynchronous message passing [1]. Each actor has a *mail address* and a *behavior*. Mail addresses may be communicated, thereby providing a dynamic communication topology. Note that the actors whose addresses an actor knows are called its *acquaintances*. In response to receiving a communication, an actor may execute the following actions:

send: asynchronously send a message to a specified actor.

create: create an actor with the specified behavior.

become: specify a new behavior (local state) to be used by the actor to respond to the next message it processes.

Specifying a parallel algorithm in terms of actors does not introduce unnecessary sequentiality. In particular, actor programs naturally represent all the parallelism in an ideal algorithm. However, the specification does not enforce a particular PDS for implementing the algorithm on a multicomputer. We install a PDS on a group of actors which implement an ideal algorithm.

For our purposes, a PDS is *fixed* or *dynamic* with respect to some set of events in computation. Events in a distributed system are partially ordered but may be mapped to a linear global time which represents the events as they may be observed by a hypothetical observer [17]. The global time is not unique. A PDS for an actor group is *fixed* with respect to a set of events E if the placement of actors in the group is fixed for all events *bounded* by E. (We define an event as *bounded* by a set of events if it must occur between some two events in the set regardless of the observer.) In intuitive terms, a fixed PDS for a computation performed by a group of actors places the actors before the computation starts and doesn't change their placement during the computation. A PDS which is not fixed is called dynamic.

Fixed PDS's are often sufficient for representing data structures where the number of data elements and their relationship during a computation is known before the computation begins: although the actors representing a static data structure may be dynamically created, the computation described by the ideal algorithm occurs after the data structure has been instantiated. By contrast, in a dynamic PDS, the placement of a group of actors performing a computation is determined during the course of a computation.

On the other hand, dynamic PDS's are especially important for parallel algorithms that use dynamic data structures because the total number of data elements performing a computation and their exact communication topology may not be fixed before the computation begins. For example, a sparse matrix structure may get modified during a computation and the total number of non-zero elements and their exact topology may not be known before the computation begins. Similarly, in a binary search tree, the structure and topology of the nodes may depend on the order in which the elements are added to the tree. For a good review of parallel algorithms which would naturally use dynamic data structures, see [32].

Dynamic PDS's may have to interact with the ongoing computation to decide the placement of newly created actors or to migrate existing ones. For example, placement decisions

for dynamic PDS's may be triggered by messages which mark the beginning of a particular phase of the algorithm or by events related to the system load such as the load on a particular node exceeding a threshold value.

Specifying a PDS

A PDS is expressed in terms of the placement (or migration) of each element in an actor group. In our model, each actor has a system actor called *meta-actor* which executes its create commands and can trap and execute migrate commands sent to it. We customize the meta-actor of an actor to place the actors it creates on specific nodes. Note that for efficiency reasons, the representation of the meta-actors for many or all actors on a given computer node may be shared in an implementation.

In our model, a data structure is represented by a group of actors. Each actor encapsulates data and has *methods* corresponding to the procedures which may manipulate the data. The procedure to be invoked is specified in the message. The group of actors have a common *group* name which may be used in conjunction with a more specific parameter to access individual actors (or a subgroup of actors) within the group (cf. [15, 12]). Given the name and a pattern, we get a particular member of the group represented by an actor address. In our case, actor groups are references to a collection of addresses and are represented by a variable reference. In case of a one dimensional array, an expression evaluating to an integer index gives us the address of a particular actor representing the corresponding element of the array. To specify a PDS for a group, we may customize the create operation on all actors which may create members of the group. Such customization may be dynamically changed.

Initial Placement

To make the discussion we specify how an array element may be placed using the form:

`create` <*array-name*> <*index*> `on` <*node-exp*>

where <*array-name*> <*index*> evaluates to a particular actor and the node expression gives the id of the multicomputer node on which the actor is to be placed. For recursive structures such as trees, the meta-actors of newly created actors may be customized to implement arbitrary PDS behaviors. This customization must itself be done by the meta-actor which is specified independently of the actor (see Figure 3).

Consider a dense matrix representation which is used by algorithms such as Gaussian Elimination, graph algorithms such as the Floyd-Worshall's algorithm for the all-pairs shortest-path problem, or domain decomposition techniques for solving Partial Differential Equations. A dense matrix may be partitioned on a parallel computer using a number of PDS's as described in [29]. Below we describe how to implement two such PDS's. The i^{th}

row of the matrix is represented as an actor group `A[i]` which is a collection of matrix elements. The matrix is represented using an actor group `Matrix` which is a collection of rows.

- *Row-wise block-striped PDS:* the i^{th} row is assigned to the $(i \text{ div } k)^{th}$ processor, where $k = n/p$. To implement the PDS, `create` is redefined in the meta-actor of the actor creating the matrix. For example, the i^{th} row of a matrix may be created by an actor using the command `create(Matrix_Row, i)`. The following code shows the customized definition of the `create` command in the meta-actor of the actors creating the rows of the matrix. The command `create(B,` *values*`) on p` executed by the meta-actor creates an actor with behavior `B` on node `p` and returns the address of the created actor to the base-actor (*values* is a list of arguments used for initialization of the newly created actor).

```
method create(behavior Matrix_Row, integer i) {
var integer location;
      location =  (i / (N / PROCS));
      return create(Matrix_row, i) on location;
   }
```

- *Row-wise cyclic-striped PDS:* the i^{th} row is assigned to the $(i \bmod p)^{th}$ processor. As above, `create` may be customized in the meta-actor of the actors creating the rows of the matrix.

```
method create(behavior Matrix_Row, integer i) {
var integer location;
      location = (i % PROCS);
      return create(Matrix_row, i) on location;
   }
```

Note that the computations related to a PDS are all implemented at the meta-actor whereas the ideal algorithm is specified as the behavior of the base-actors. This provides the modular separation of the specification of PDS's from the ideal algorithms. Installation of the appropriate PDS before or during a computation of the ideal algorithm provides the composition of the two specifications.

Migration

To change the placement of an actor during the course of a computation, *migration* is necessary. The migration of an actor may be triggered by sending a `migrate(<`*node-expression*`>)` message to the actor. This message is trapped by the meta-actor and executed by moving the actor to the processor with address given by *node-expression*. The `migrate` message itself may be sent by other meta-actors, e.g., a group of actors may form a tree data structure and the parent of an actor may send the `migrate` message to its children.

There are two potential reasons to use migration. First, different phases of an algorithm (or an application) operating on the same actor group may be implemented more efficiently using different PDS's. Such changes may be specified statically but triggered dynamically when a given component reaches a particular phase. Second, the irregular nature of a computation may mean that the placement should be determined by current conditions. In particular, the dynamic behavior of the computation may be used in heuristics which determine placement. Adaptive load balancing or diffusion scheduling are two examples of such strategies.

For example, the computation of a linear equation solution phase of an algorithm may be started by a message `iteration(0)` sent to actor `A[0]` which represents the first row of the matrix. The event that triggers the migration of actor `A[0]`, an element in a group of actors that form the matrix data structure, may be specified by trapping a message of the form `A[0].iteration(0)` in the meta-actor. An appropriately customized meta-actor will migrate the actor and propagate a move message to the children. After migration, the actor's meta-level will release the `A[0].iteration(0)` message to its base actor `A[0]`. For more complex algorithms, the meta-actor may cause migration of other actors in a group to be recursively triggered. This may be achieved, for example, by customizing the meta-actor of an actor to compute the new placement in each meta-actor in response to some specific message (we give an example of this in the next section).

Dynamic Installation of PDS

Composition of a PDS with an ideal algorithm requires the user to specify when a PDS must be triggered so that actors are created or migrated accordingly. We allow external pattern based specification of events which trigger a PDS mechanisms to specify abstract (coordination patterns are described in [19]). For example, a pattern may specify a load imbalance in the system, or it may represent invocation of a method that starts the computation of a new phase of an algorithm, which triggers migration of actors in a group.

In order to maintain modularity, such triggering may not be based on the encapsulated state of the computational objects implementing a specific implementation but may rely on their interface and invocation history. The invocation history represents external messages sent to actors in the group. In our methodology such triggering is done by actors called *managers*. Managers may interact with system level actors, for example, to determine the load, and may examine messages sent to actors they are managing. To install a PDS, they may modify a meta-actor's behavior(see [3] for a discussion of customizing meta-actors).

4 A Detailed Example: Sparse Cholesky Factorization

In this section we illustrate how our methodology may be used to implement PDS's for a sparse matrix representation. The ideal algorithm discussed is the parallel sparse Cholesky

Factorization algorithm [23, 38]. For a given symmetric positive definite matrix A of size $n \times n$ the Cholesky Factorization algorithm computes a lower triangular matrix L, of size $n \times n$ such that $A = LL^T$ [21]. PDS's for parallel sparse Cholesky Factorization are discussed in [23, 20].

The given sparse matrix A is represented using the edge list representation of a graph. Each column of the original matrix is represented as an actor. The details of the sparse Cholesky Factorization algorithm can be obtained from [23]. Before computing the elements of matrix L, the algorithm constructs an elimination tree [20] which contains dependence information between the columns of the sparse matrix A. If node i is a descendant of node j in the elimination tree then column i of L must be computed before column j. The actors representing the columns of the sparse matrix extend their communication topology to form the elimination tree. The information related to column dependencies available in the elimination tree can be used to assign the columns to the processors so as to minimize communication and balance load. Several PDS's for sparse matrices based on elimination trees may composed with ideal algorithms that use such a data representation.

Figures 4 describes the meta-architecture for installation of the PDS's (the numbers corresponding to messages in the following text refer to the figure). The migration of the actors in the group to satisfy a new PDS requires meta-actor to communicate with the meta-actors of the two children of its base node (called `lchild` and `rchild`) but does not otherwise depend on the specific representation of the ideal algorithm in the tree nodes. The behavior to implement the PDS is sent by a manager to the actor which when applied by the meta-actor results in both the installation of a customized meta-actor and the PDS being propagated down the tree. The placement strategy specified by the new PDS will be triggered as described by the PDS.

Let `n1` be the root node of the elimination tree and `n2`, `n3` be its children. The computation which decides the new placements for each of the actors is triggered by the arrival of `start_cholesky` message at the root node (1). Specifically, the meta-actor of the root node starts the computation of the destination processors for the various nodes of the tree by propagating messages down the tree (2,3). This propagation carries the PDS to compute the placement of the children. When the root node has migrated, it invokes the method `start_cholesky` at the root (4).

Next we describe details of two of the PDS's based on the elimination tree and show how we implement them using our methodology.

Subtree-to-subcube PDS

A subcube represents a set of p processors where p is a power of two. Starting at the root of a subtree there may be a chain of nodes which is mapped in a wrap-around fashion to the corresponding subcube. If the tree divides into two subtrees, the subcube of processors is divided into two subcubes and each subtree is mapped to one subcube. If a subtree is mapped to a cube containing a single processor, all the nodes of that subtree are mapped to that processor.

This algorithm computes the placement of the tree nodes in a single pass from the root node to the leaves. A subtree of the elimination tree is mapped to an ordered set of processors called a `subcube`. When the meta-actor of the root node receives the `start_cholesky` signal it triggers the sparse-wrap PDS by sending the message `move(p,cube)` to the meta-actor of the root node (here `cube` is the set of all processors in the given architecture and `p` is one of the processors belonging to `cube`). A tree node meta-actor which receives a message `move(p,subcube)` performs two activities. It first propagates the `move` signal to its children with possibly different values of the parameters and then migrates its base-actor to the processor `p`. Next we describe how the `move` signal is propagated to its children.

If a tree node has only one child, it is a part of the initial chain of nodes which starts at the root of a subtree. Such a tree node, upon receipt of a message `move(p,subcube)` sends a message `move(p',subcube)` to its children where `p'` is the successor of processor `p` in the subcube. If the tree divides into two subtrees, the subcube is divided into two subcubes `subcube1` and `subcube2` and messages `move(p1, subcube1)` and `move(p2, subcube2)` are sent to the left and right child respectively (`p1` and `p2` are processors that belong to `subcube1` and `subcube2` respectively). If a node receives a message `move(p,subcube)` where `subcube` consists of only one processor `p`, this message is propagated down to every node in the subtree so that the whole subtree is mapped to the processor `p`. The algorithm takes time $O(h)$ where h is the height of the tree.

Sparse-Wrap PDS

The sparse-wrap PDS assigns all the leaves of the elimination tree in a wrap fashion. Assuming these nodes are removed from the elimination tree, the next set of leaves is assigned, and so on.

The execution of the sparse-wrap PDS is started by sending the `start_count_leaves` message to the meta-actor of the root node. The first phase of the PDS results in the computation of the number of leaves in the subtrees pointed at by the left and right child of each node. The next phase of the PDS is started by the root node and assigns a unique integer identifier to each leaf. If there are n leaves in the elimination tree, the identifier i for a leaf has a value $0 \leq i < n$. The sparse-wrap mapping of a leaf with identifier i is given by (i mod P) if there are P processors. When the mapping of a leaf node is computed it is migrated to its destination processor.

Note that the computation of the mapping of the leaves of an elimination tree of height h takes $O(h)$ time in parallel. Computing the mapping of all the nodes, requires h such passes which remove the current set of leaves and compute the mapping for the next set of leaves. The execution of the different passes of this algorithm can be pipelined so that the overall time taken is $O(h)$ (with unbounded resources).

5 Implementation

The PDS's for the dense matrix and the sparse matrix structures, and the ideal algorithms described in this paper were implemented using an actor kernel called *Broadway*. The kernel provides a minimal set of primitive operations that may be used to efficiently construct more complex abstractions. Broadway is built as a C++ library which facilitates the specialization and customization of system components. In particular, Broadway may be used to support multiple actor languages. The example programs presented in the paper are written in *Screed*, a high level actor language that generates code for Broadway.

Broadway has been designed for portability – all machine dependent operations are implemented in a single platform module which handles actor creation and interprocess communication. Broadway performs its own scheduling. Broadway currently runs on a network of DEC and Sun workstations. The platform module uses sockets for internode communication. For more details on Broadway see [34].

We present performance results of an implementation of two commonly used benchmarks written in Screed and executed on a network of Sun Sparc stations. It should be noted that the purpose of the paper is to present a new methodology to simplify programming scalable parallel software rather than to report on an optimized system. Thus the performance results are only to document the state of the implementation. Broadway has been developed as a platform for prototyping high-level programming constructs for actor languages rather than as a tool for scientific computing.

In fact, on standard benchmarks, the absolute performance of the current implementation is not competitive with implementations of other more restrictive programming models. For example, the performance figures for Mentat [22] are several fold better. However, it should be noted that the network we used in our experiments was not a dedicated network for running Broadway but was shared with other users. More critically, our methodology requires the actor program to be specified as a fine grained computation so that actors can be grouped in different ways to implement different PDS's. Creation of a large number of actors results in significant overhead. The overhead may be reduced by using a number of compiler optimization techniques we are studying (e.g., see [6]) – however, it is too early to tell how much of the overhead will remain after such optimizations.

The benchmarks we report are Gaussian elimination and matrix multiplication since results are available in literature for executing these benchmarks on other systems. The speedups obtained for performing Gaussian elimination on a 256×256 matrix are as follows.

Processors	Speedup
2	1.9
4	3.6
8	5.7

Speedups obtained for performing matrix multiplication on a 256×256 matrix are as follows.

Processors	Speedup
2	1.7
4	3.2
8	5.8

The partitioning strategy used for both the algorithms is row-wise cyclic-striped. The sequential times for the Gaussian Elimination and Matrix multiplication obtained by running Broadway on a uniprocessor case are 116 seconds and 170 seconds, respectively. Sequential, optimized C programs for Gaussian Elimination and Matrix Multiplication on a matrix of size 256×256 take 11 seconds and 37 seconds respectively.

6 Discussion

In this section we present a brief review of some of the existing programming languages that provide support for data partitioning and distribution of parallel programs. Support for partitioning and distributing arrays is provided in languages such as Kali [11], Vienna Fortran [14] and Fortran D [28]. The languages Kali and Vienna Fortran allow the programmer to declare a processor array and specify distribution of each dimension of a data array onto the processor array. In Fortran D, data distribution is specified using abstract structures called decompositions which provide a frame of reference for inter-array alignment. The ALIGN statement is used to map arrays onto decompositions. The decompositions are mapped to the physical machine by using the DISTRIBUTE statement. Both Vienna Fortran and Fortran D allow certain intrinsic distribution functions such as BLOCK, CYCLIC, and BLOCK-CYCLIC.

Kali and PARTI [18] support sparse and unstructured computations using distributed arrays accessed using indirection. They transform a sequential loop into two constructs namely, inspector and the executor. During program execution, the inspector loop examines the data references made by a processor and calculates what remote data needs to be accessed and where it is stored locally. The executor loop uses the information from the inspector to implement the actual computation. Vienna Fortran and Fortran D allow irregular distributions by specifying the distribution function as an integer-valued mapping array of the same shape and size as the data array. Fortran D also supports dynamic alteration of the distribution of an array since ALIGN and DISTRIBUTE are executable statements. The constructs provided for specifying PDS's in languages such as Vienna Fortran, PARTI and Fortran D do not utilize the features provided by object-oriented technology such as encapsulation and inheritance. There is no support for dynamic allocation of storage and therefore dynamic data structures can only be mimicked by allocating large arrays statically.

A number of approaches extend the sequential object-oriented language C++ with constructs for parallel computing. In particular, Chandy and Kesselman have developed Compositional C++ (CC++) which extends C++ with parallel programming constructs such as `par`, `parfor` and `spawn` [13]. CC++ allows multiple threads per object and objects do not necessarily represent computational agents that compute concurrently. Synchronization

between concurrently executing components uses single assignment objects called `sync` objects. Another effort whose goal is to introduce concurrency in C++ is μC++ [10]. μC++, adds features such as coroutines, monitors and threads to C++ and uses a *single-memory* model: communication and synchronization between concurrently executing threads is through shared variables and monitors. A number of actor based languages have been developed and used for parallel and distributed programming (for example, Rosette [36], Acore [31], Cantor[8], HAL [24], ABCL [35], Concurrent Aggregates [16], ACT++ [26] and Charm [27]). Other concurrent object-oriented programming languages using providing mechanisms for defining groups of objects is PC++ [30], and Mentat [22] which provides an efficient runtime system. However, none of these concurrent object-oriented programming languages or systems provides support for the modular specification of PDS's.

In this paper, we presented a methodology for programming concurrent computers which allows separate specification of an ideal algorithm and the PDS's. Our methodology allows expression of PDS's for static as well as dynamic data structures: specifically we can define representative prototypes which encapsulate details of the architecture dependent PDS's for a groups of actors corresponding to given abstract data type. Moreover the usual object-oriented methods can be applied to organizing PDS's: an actor's inheritance hierarchy may be used to share meta-actors implementing a PDS; alternately delegation can be used. Finally, it should be noted that the efficiency with which a flexible system such as ours can be implemented remains open to question. We are investigating compiler optimizations for generating efficient code for the high level language constructs we use (see, for example, [6]).

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III, IFIP Transactions.* Elsevier Science Publisher, 1993.

[4] G. Agha, C. Houck, and R. Panwar. Distributed execution of actor systems. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 1–17. Springer-Verlag, 1992. Lecture Notes in Computer Science 589.

[5] G. Agha and R. Panwar. An actor-based framework for heterogeneous systems. In *Proceedings of the Workshop on Heterogeneous Processing*, pages 35–42. IEEE, 1991.

[6] G. Agha and K. Wooyoung. Compilation of a highly parallel actor-based language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing.* Yale University, Springer-Verlag, 1992. LNCS, to be published.

[7] G. A. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, May 1993.

[8] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–23, August 1988.

[9] J. Bruno and P. R. Cappello. Implementing the beam and warming method on the hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 1073–1087, Pasadena, CA, January 1988.

[10] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zaranke. μC++: Concurrency in the object-oriented language C++. *Software – Practice and Experience*, 22(2):137–172, February 1992.

[11] P. Mehrotra C. Koelbel. Compiling global name-space parallel loops for distibuted execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[12] C. J. Callsen and G. A. Agha. Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, 1994. (to appear).

[13] K. M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1993.

[14] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna fortran - a fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, 1991.

[15] A. Chien. *Concurrent Aggregates: An Object-Oriented Language for Fine-Grained Message-Passing Machines*. PhD thesis, MIT, 1990.

[16] A. Chien. Supporting modularity in highly-parallel programs. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1993.

[17] W. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[18] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*. Elsevier Science Publishers, 1992.

[19] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, July 1993. LNCS 627.

[20] G. A. Geist and E. Ng. Task scheduling for parallel sparse cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[21] G. Golub and C. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1983.

[22] A. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic object-oriented parallel processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):33–47, May 1993.

[23] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, September 1991.

[24] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of th 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[25] L. H. Jamieson. Characterizing parallel algorithms. In R. J. Douglass L.H. Jamieson, D.B. Gannon, editor, *The Characteristics of Parallel Algorithms*, pages 65–100. MIT Press, 1987.

[26] D. Kafura. Concurrent object-oriented real-time systems research. *SIGPLAN Notices*, 24(4):203–205, April 1989. (Proceedings of the Workshop on Object-Based Concurrent Programming, 1988).

[27] L. Kale. *The CHARM(3.0) Programming Language Manual.* University of Illinois, October 1991.

[28] Ken Kennedy and Ulrich Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report Rice COMP TR91-155, Rice University, 1991.

[29] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings Publishing Company, Inc., 1994.

[30] Jenq Kuen Lee and Dennis Gannon. Object-oriented parallel programming experiments and results. In *Proceedings Supercomputing 91*, pages 273–282, 1991.

[31] Carl Manning. Acore: The design of a core actor language and its compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[32] P. Mehrotra, J. Saltz, and R. Voigt, editors. *Unstructured Scientific Computation on Scalable Multiprocessors.* MIT Press, Cambridge, Massachussets, 1992.

[33] P. Porta. Implicit finite-difference simulation of an internal flow on hypercube. Research Report YALEU/DCS/RR-594, Yale University, January 1988.

[34] D. C. Sturman. Fault-adaptation for systems in unpredictable environments. Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[35] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 218–228, May 1993.

[36] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in carnot. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2), May 1993.

[37] E. S. Wu, R. Wesley, and D. Calahan. Performance analysis and projections for a massively-parallel Navier-Stokes implementation. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[38] E. Zmijewski and J. R. Gilbert. A parallel numerical algorithm for large sparse symbolic and numeric cholesky factorization on a multiprocessor. Technical Report TR 86-733, Department of Computer Science, Cornell University, Ithaca, New York, February 1986.

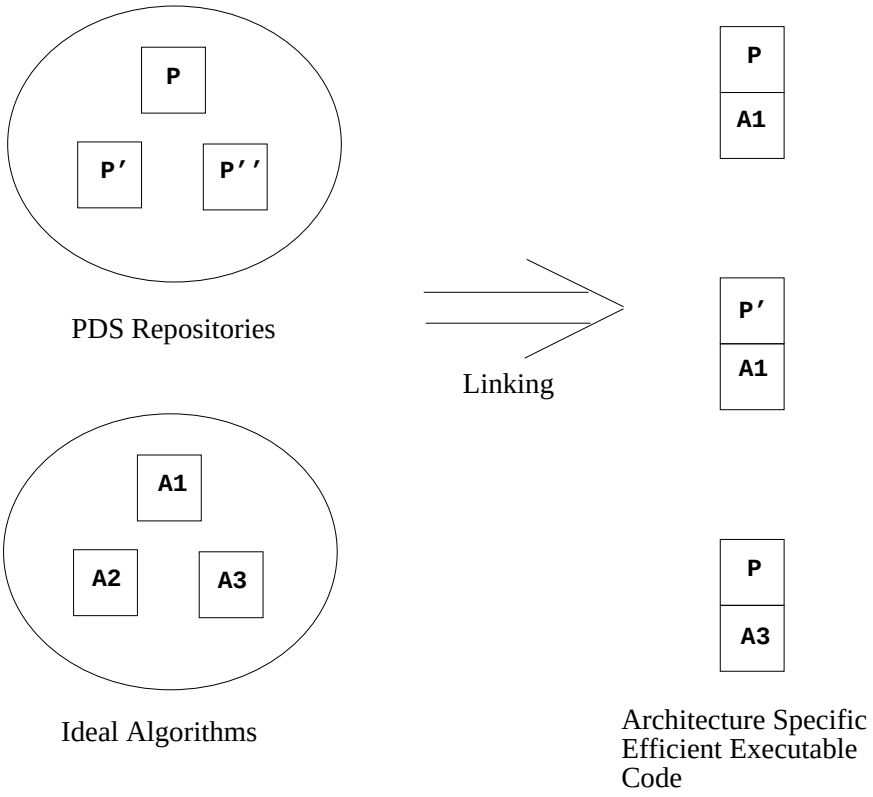

Figure 1: Combining ideal algorithm specification with a PDS to obtain architecture specific, efficient executable code.

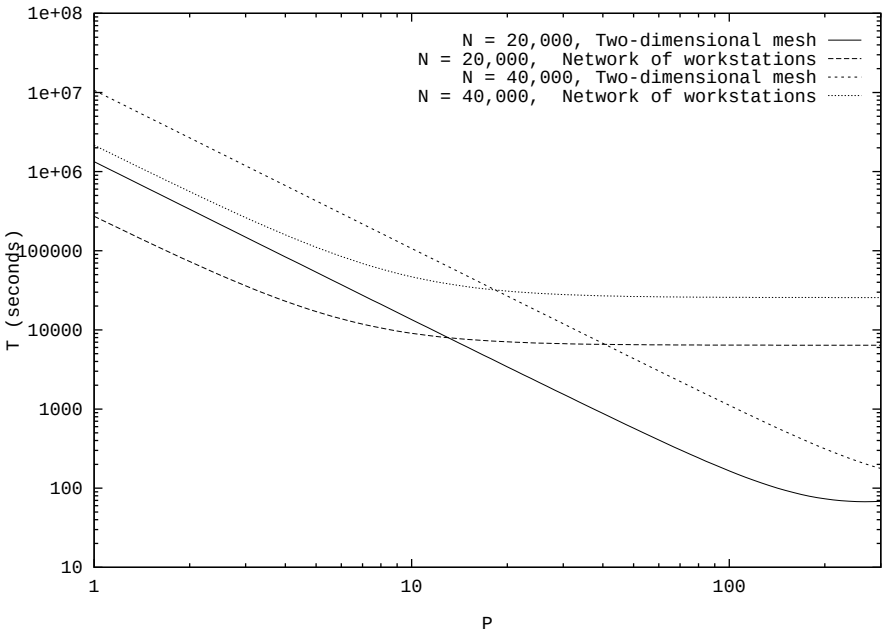

Figure 2: Time taken by Cholesky decomposition algorithm (T) on two-dimensional mesh and a network of workstations with varying number of processors (P) and matrix size N $\times$ N. The ratio of the communication time to the computation time for the network of workstations is assumed to be 20 times the ratio for mesh architecture. The workstations are assumed 5 times faster than the processors on the mesh.

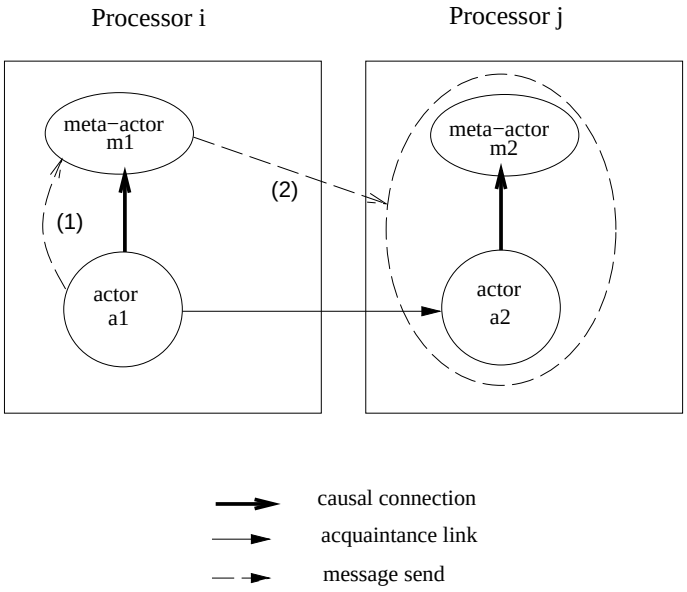

Figure 3: Meta-architecture for implementing a PDS. The execution of `create(B)` command in actor `a1` causes a `create(B)` request to be sent to its meta-actor `m1` (2). `m1` creates the actor `a2` and `m2`, the meta-actor of `a2` using the behavior of the given PDS (2).

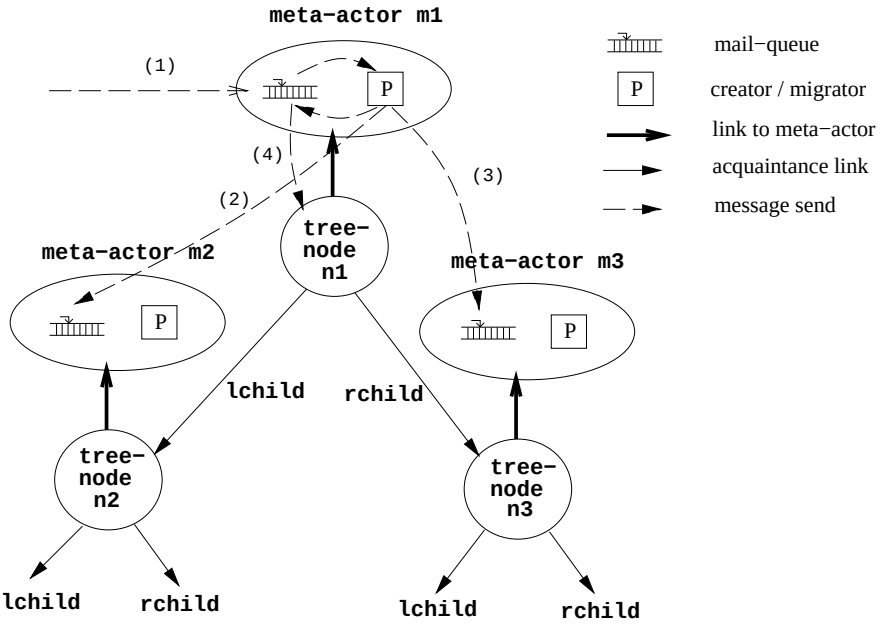

Figure 4: Meta-architecture for installing PDS's for sparse matrix representation used for Cholesky Factorization.