An Actor-Based Framework for Heterogeneous Computing Systems*

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

Abstract

This paper discusses a framework for supporting heterogeneous computing systems (HCS). HCS *are physically distributed, potentially complex and evolving systems.* HCS *consist of a number of computers and interconnections which may have differing language and system support, and sometimes distinct computational (architectural) models. By providing a large number of computational resources to the user,* HCS *have the potential to allow highly efficient execution of ultra large-scale applications. The paper provides a framework for addressing some of the significant problems in interoperability and resource management which result from the heterogeneity in* HCS.

1 INTRODUCTION

A number of distinct parallel and distributed architectures and interconnection networks have been designed. Although building a "general-purpose" architecture has been a key design goal for most computer architects, the reality is that many algorithms have different structural characteristics which make them more suitable for execution on very specific kinds of architectures. Some algorithms require very close interaction between processors, making architectures with high processor connectivity suitable. Other algorithms may be partitioned into components which have strong locality properties. In this case, distributed memory architectures are appropriate. A third group of algorithms exhibits a high degree of data parallelism.

For example, algorithms for solving a sparse system of linear equations partition the system of equations into smaller blocks, and are therefore suitable for a medium grained machine such as a mesh. On the other hand, algorithms for image processing have a regular structure with very high data parallelism; this allows efficient execution on SIMD machines such as CM2.

Large-scale applications consist of many components, each of which may be suitable for execution on a different kind of architecture. By developing heterogeneous computing systems (HCS) we can efficiently support such applications. For example, consider an application that can be decomposed into two components, one of which requires very frequent processor interaction whereas the other is data parallel. If this application is executed on a shared memory machine, data parallelism in the second component may not be effectively used. On the other hand, if CM2 were to be used, the first component of the problem would not be implemented efficiently. If several architectures with different characteristics are connected on a high speed network, it may be possible to execute various components of an application on different architectures.

There are a number of difficulties which must be addressed in order to realize the promise of HCS. This paper discusses two of them. First, the absence of a uniform language and communication interface on different systems makes programming HCS difficult. Thus, the programmer who writes applications for HCS may have to program each component of the application using the the language support for the architecture on which the component is to be executed. In particular, if a component is suitable for more than one architecture, then the programmer may have to provide the code for all architectures on which the component may be executed. To address this proble, we propose an actor-based linguistic interface model for interoperability in HCS

Actors provide behavior abstractions which unify abstract data types and higher-order functions. As a consequence, modularity and reusability are supported in this linguistic framework. Experience in practical distributed systems has shown the power of actor languages. [1]

*The authors thank P. Vaidya for discussions on branch and bound techniques, B. Ramkumar for discussions on resource management and D. Sturman for suggestions on the manuscript. The research described has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), 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).

[1] For example, a group of programmers using Rosette (an actor language developed at Microelectronics and Computer Technology Consortium (MCC) in collaboration with the first

A second problem which needs to be addressed in order to make HCS viable results from the fact that the von Neumann architecture no longer provides a universal model of computation. In particular, this implies that conventional measures of space-time complexity are insufficient to model resource requirements. HCS provide a wide variety of resources; depending on how these resources are used, varying degrees of performance may be achieved. Measures of an algorithm's complexity and scalability may provide a quantitative basis which helps select appropriate resources to use in executing it. In this paper, we review several metrics for evaluating the utilization of computational resources and discuss how they may be used in building a resource management architecture.

The outline of this paper is as follows. We first provide examples to motivate the need for heterogeneous processing of applications. Then we discuss the problem of providing linguistic support for HCS. In particular, we describe a reflective model of actors and show how it can be used to meet design requirements for interoperability in HCS. Note that reflection allows the underlying resources to be represented and thus provides a mechanism for affecting resource utilitzation. The second half of the paper describes formal metrics which need to be incorporated in a resource management architecture for HCS and illustrates their significance by means of specific examples. Finally, we discuss the use of meta-programming for building reusable orchestration tools.

MOTIVATING EXAMPLES

A number of large-scale applications are composed of smaller computations each of which may require a different kind of architecture. Solving such applications on one architecture leads to poor performance since certain components of the application may not be suitable for that architecture, leading to their inefficient implementation. We describe two such examples: applications in computer vision and preconditioned conjugate gradient method for solving linear systems of equations, a numerical computation.

Computer Vision:

Processing images and making intelligent decisions based on the data requires different levels of processing: low-level, intermediate-level and high-level [7]. Every level of processing has a different set of characteristics and is suitable for a different kind of parallel machine.

- Low-level vision tasks such as Hough transform computation, histogram generation and template matching require computations corresponding to each pixel of the image. These computations are highly regular and the same computation is performed on each pixel of the image. Thus SIMD architectures, such as the Connection Machine, are well suited for low-level vision tasks.

- Intermediate-level processing reduces the low-level information to a form that can be easily processed by high-level processors. Examples of intermediate level processing are segmentation, computation of connected components, syntactic pattern recognition etc. These computations are not as regular as the low-level vision computations and the parallelism cannot be extracted very easily. The parallel tasks generated here have a larger granularity and hence machines such as hypercubes or shared-memory machines are more suited.

- High-level processing requires making intelligent decisions based on image data and the algorithms are not very structured. For example image data is transformed into information that needs symbolic processing. Such applications may require higher connectivity or shared memory machines for processing although certain specific algorithms have been developed and implemented on the Connection Machine [7].

Preconditioned Conjugate Gradient Algorithm:

This problem [10] requires various steps that have very different requirements of the architecture. Assume the input to the algorithm to be a large sparse system of linear equations.

- *Developing the Preconditioner:* This operation can be very unstructured and may require certain graph algorithms to compute the preconditioner from the input matrix. Thus it is best suited to a fast sequential processor or a shared memory multiprocessor.

  Once the preconditioner is obtained, the algorithm performs several iterations where each iteration is composed of several components including:

  - *Solution of Loosely Connected Blocks of Equations:* This operation involves the solution of several blocks of equations that are either loosely coupled or independent of each other. Thus, the connectivity required for solving such an equation is not high. The grain size of the computation depends on the number of blocks into which the system of equations is divided. Such a computation is suited for a medium grain multiprocessor which need not have very high connectivity.
  - *Solution of a Dense System of Equations:* This is a structured computation that requires high communication. Thus either vec-

author) was able to develop front-end to back-end switching package which supported interoperability in heterogeneous distributed database applications. The implementation done for the ESQL Access Group, a consortium of vendors, provided efficiency comparable to implementations done directly using low level network communication protocols. However, the Rosette implementation took a small team of three programmers about six weeks, while competing implementations took three to four times as many programmers upto a year or more.

tor processing or a strongly connected architecture, such as a hypercube, is suitable.

We analyze an application with several components that are executable on parallel machines. Assume there are n different components such the i^{th} component is suitable for a machine M_i. It is possible that $M_i = M_j$ even though $i \neq j$. Let $T(C_i, M_j)$ be the time required for execution of component C_i on machine M_j and T_{seq} be the time required for sequential execution of the application. The speedup obtained by executing all components of the application on one parallel machine M_k is:

$$S_{PM} = \frac{T_{seq}}{\sum_{i=1}^{n} T(C_i, M_k)}$$

Instead, if an HCS is used and every component of the application can be executed on the machine suitable for it, the speedup obtained is:

$$S_{HCS} = \frac{T_{seq}}{\sum_{i=1}^{n} (C_i, M_i) + T_{NO}},$$

where T_{NO} is the network overhead i.e., the total time spent in transferring data from one machine to another over the network. It is advantageous to use the HCS only if $S_{HCS} > S_{PM}$ which holds if,

$$\sum_{i=1}^{n} (C_i, M_k) > \sum_{i=1}^{n} T(C_i, M_i) + T_{NO}.$$

Thus, we need to ensure that the network overhead does not offset the advantage caused by efficient execution of the components on different machines. In the rest of the paper we assume the different machines of an HCS to be on a very high-speed network. In the next section we discuss the features required in the language support for such HCS.

2 LANGUAGE SUPPORT FOR HCS

Heterogeneous Computing systems are complex, constantly changing dynamic systems and programming such systems can be highly complicated without proper language support. The language should provide enough levels of abstraction to hide unnecessary implementation details of an algorithm without restricting the programmers ability to manage system resources for efficient execution. For an HCS, such parameters include the selection of the appropriate architecture for a particular component of an application, and the management of resources such as the number of processors. Below are some design requirements for providing language support for HCS. We will show how the actor model may be used to address these requirements.

- *Inherent Concurrency:* The prerequisite for any language used for HCS is to allow concurrency. It should be possible to create several processes which can be executed in parallel. The creation of processes should be dynamic. Communication protocols are necessary to allow interaction between applications running on two different machines on the network. The language should provide some abstractions which encapsulate the details of such a protocol.

- *Modeling Dynamic Systems:* A heterogenous system may keep changing constantly as new machines are added and old machines are removed. Thus a language which supports a dynamic topology of processes is required. The static topology required by a language such as CSP is highly restrictive. Functional languages are not suitable for modeling resources since they cannot model the state of a system. In addition, as the number of resources in an HCS increases, the number of failures increases and fault tolerance becomes an important issue.

- *Machine Independence:* The language should be independent of the machines on which the program is executed. If a machine independent language is not available, the programmer may have to use a different language for every component of an application. If one component of an application can be executed efficiently on more than one machine, it should be possible for the system to choose a suitable machine based on parameters such as availability and load status. Thus the lack of a machine independent language implies that a programmer would have to provide several versions of a component code, each version suitable for a different architecture.

- *Choosing the Machine Configuration for a Problem:* The suitability of an architecture to a given problem depends on several factors such as the structure of the problem, the available parallelism, the communication overhead involved, etc. It is difficult to analyze a program statically and find out the possible target architectures suitable for it. The language should allow the user to give various metrics associated with the application which should help the system make resource management decisions.

- *Process Placement:* For certain applications, it is useful to allow the programmer to explicitly allocate resources and control the data and task placement. The user may know the most efficient way of executing the program and may be the best judge of how data should be distributed across various processors.

- *Modularity of Code:* The language should allow modularity in design and allow a programmer to separate the various design considerations. For example, the code for resource management and explicit task placement (if any) should be separate from the code of the problem itself. Because it is simpler to reason about functional code, the code for the computation itself may be implemented in a functional style. On the other hand,

the code for resource management is easier to express if one can represent states of a computation. The object model allows us to integrate the two styles.

2.1 THE ACTOR MODEL

Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. Each actor has a *mail address*, and a *behavior*. An actor's *acquaintances* are all of the actors whose mail addresses it knows. In order to abstract over processor speeds and allow adaptive routing, preservation of message order is not guaranteed. Addresses may be communicated, providing a dynamic topology. New actors may be dynamically created – providing extensibility.

State change in actors is specified using replacement behaviors. Each time an actor processes a communication, it also computes its behavior in response to the next communication it may process. The replacement behavior for a purely functional actor is identical to the original behavior; in general it may change. The change in the behavior of an actor may represent a simple change of state variables, such as change in the balance of an account, or it may represent changes in the operations (methods) which are carried out in response to messages. For example, suppose a bank account actor accepts a withdrawal request. In response, it will compute a new balance which will be used to process the next message.

The concept of actors was originally proposed by Hewitt [11]. The actor model was formally characterized by means of power domain semantics [8], by a transition system [1], and by Colored Petri Nets [13]. Complexity measures for actor programs have been studied [6]. The model has also been proposed as a basis for multiparadigm programming [2] and has been used as a programming model for multicomputers [5, 9].

2.2 LINGUISTIC EXPRESSIVENESS FOR INTEROPERABILITY

In systems with shared memory, synchronization constructs based on shared variables, such as semaphores and monitors, are used. These constructs allow various processes to synchronize and share information. However applications running on two different machines on a network have to synchronize by sending messages over the network. Thus message passing primitives such as `send` and `receive` are used for communicating. If the `send` primitive does not cause a process to block until the message reaches the receiver, the communication is said to be *asynchronous*. If the sender blocks until the other process is ready to receive the message, the communication is said to be *synchronous*.

There are other higher level protocols built using these primitives e.g., remote procedure calls (RPCs) and rendezvous. In a remote procedure call, the servicing process is declared as a procedure. When a call is made, the procedure may get executed on a remote processor. The term rendezvous refers to a form of synchronous communication in which two processes get to a designated point. The process that reaches the point first waits for the other process. Another form of synchronization is barrier synchronization in which several processes meet at a point. The processes that arrive early wait for the slow processes. As soon as the last process reaches the barrier point, all the processes continue execution.

The actors model uses message passing for communicating between tasks. The underlying network is assumed to be asynchronous. All the other protocols can be implemented using asynchronous message passing. For example synchronous communication can be modelled using asynchronous message passing by forcing the sender to keep waiting for the reply.

Assume that `P1, P2, P3` are actors with behavior `sync-proc`. These processes know the mail address of a `barrier-actor` called `B1`. All the processes participating in the barrier synchronization send a `b-sync` message to their `barrier` acquaintance as soon as they reach the synchronization point.

```
(defActor sync-proc (slots& barrier)
    (method (send-sync)
       (send b-sync barrier))
    (method (other-computation)
      ...   ))
```

The `barrier-actor` allows three processes to synchronize at a point. It has three acquaintances, the integers `message-rec` and `barrier-size` and a list of process ids `proc-list`. The acquaintance `barrier-size` contains the number of processes that can synchronize at the barrier. Thus `barrier-size` receives its initial value at the creation of the `barrier-actor`. The acquaintance `message-rec` acts as a state variable that stores the number of processes that have sent the synchronization message `b-sync` to the `barrier-actor` so far. This variable is initialized to zero. Finally, `proc-list` contains the list of ids of all the processes that have sent `b-sync` message to `barrier-sync` so far. This list is necessary to broadcast the `continue` message to the synchronizing processes once the `b-sync` message is received from all of them.

```
(defActor barrier-actor
    (slots& message-rec barrier-size
proc-list)
    (method (b-sync proc-id)
       (seq
          (increment message-rec)
          (add-proc-list proc-id)
          (if (= message-rec barrier-size)
              (seq
                  (send continue
                      (new continuation)
proc-list)
                  (become (barrier-actor
                       0 barrier-size
[])))))))
```

The procedure `increment` increments the value of the variable `message-rec` by one, thereby changing the state of the `barrier-actor`. The procedure `add-proc-list` adds the mail address of the `sync-proc` actor that has sent the `b-sync` message

to the `proc-list`. The procedure `broadcast` sends the `continue` message to all the processes in the list `proc-list`.

Once barrier synchronization is achieved, the `barrier-actor` sends a message to a `continuation` actor. The `continuation` actor decides what further action has to be taken. The simplest action is to send a message to all the synchronizing processes to continue.

```
(defActor continuation
     (method (continue proc-list)
        (broadcast proc-list
other-computation)))
```

The `root` actor shown below creates the barrier actor and the process actors that use the barrier for synchronization.

```
(defActor root
    (method (test)
        (let [[B1 (new Barrier 0 3 [])]]
           (send-sync (new sync-proc B1))
           (send-sync (new sync-proc B1))
           (send-sync (new sync-proc
B1)))))
```

In fact high-level language primitives can be built in the actor language that allow the programmer to write concurrent applications without explicitly setting up any communication protocols or synchronization primitives between the tasks. We show an application, the function `concurrent-map` to illustrate how high level language constructs create processes to perform computations in parallel and also hide the communication details from the user.

The `map` function in lisp takes a function and a list as arguments and returns a list formed by applying the function to every element of the list. The `concurrent-map` function is a concurrent implementation of the `map` function. It recursively divides the input list into two halves and applies the `concurrent-map` functions to the smaller list. The recursive division of the lists continues until single element lists are obtained. The input function is applied to the elements of these lists and the results concatenated to form a list which is returned. Thus several sub-tasks are created in a tree like fashion. The results are returned to the calling actor also in a tree like fashion.

```
(defProc concurrent-map (list Function)
  (if  (> (size list) 1)
    (concatenate
      (concurrent-map ((first-half list)
Function))
      (concurrent-map ((right-half list)
Function)))
          (list (Function (car list)))))
```

The execution of the `concurrent-map` function is illustrated in the Figure 2.

In the next section we discuss the features that need to be added to the language to allow efficient resource management.

3 RESOURCE MANAGEMENT ARCHITECTURE

It is difficult to decide statically, if a given task can be efficiently executed on a given architecture. One way of choosing suitable architectures for a given piece of code is to let the user provide some important metrics related to the algorithms. These metrics should enable the system to analyze the efficiency of execution of the application on a given architecture. At compile time, executable code should be generated only for machines for which the efficiency of execution is not very low. At run time the system should choose a machine based on efficiency criteria and also on availability of machines. Below are some metrics which may enable a resource management system to select a suitable machine for a given algorithm.

- *Concurrency Index:* The Concurrency Index (CI) [3] gives a measure of the amount of parallelism available in a given computation. The higher the value of CI, the more is the parallelism available in the computation.

- *Grain Size:* This metric gives the amount of computation performed before a message is sent out. Machines such as the Connection Machine have a large number of small processors and are suitable for computations with small grain size. Machines with fewer but more powerful processors are preferred for computations with large grain size.

- *Communication/Computation Ratio:* This metric is a measure of the amount of data communicated after performing a unit computation. A high communication to computation ratio indicates a communication intensive job which may not be executed efficiently if a very large number of processors are used. For example, in the Cholesky Decomposition algorithm [4] the communication/computation ratio is quite high, indicating that the computation will be slow if run on a machine, such as a one-dimensional array, which has a high latency and low bandwidth.

- *Scalability Measure:* This measure tries to combine the information in the above two measures. One scalability measure *isoefficiency* relates the problem size to the number of processes necessary for linear speedup. If the computation size needs to grow as $f(p)$, where p is the number of processors, to maintain an efficiency E, then $f(p)$ is defined to be the **isoefficiency function** for efficiency E [14]. Isoefficiency functions for quick-sort implementations are presented for the two-dimensional mesh architecture in [14]. A naive implementation stores the entire list in one processor; it partitions the list into two sublists, hands out one of the list to a free processor and keeps the other sublist. This process continues recursively until all processors are busy. This algorithm has a very poor scalability and the isoefficiency function is shown to be:

$$W = \Theta(2^{kp} \times p)$$

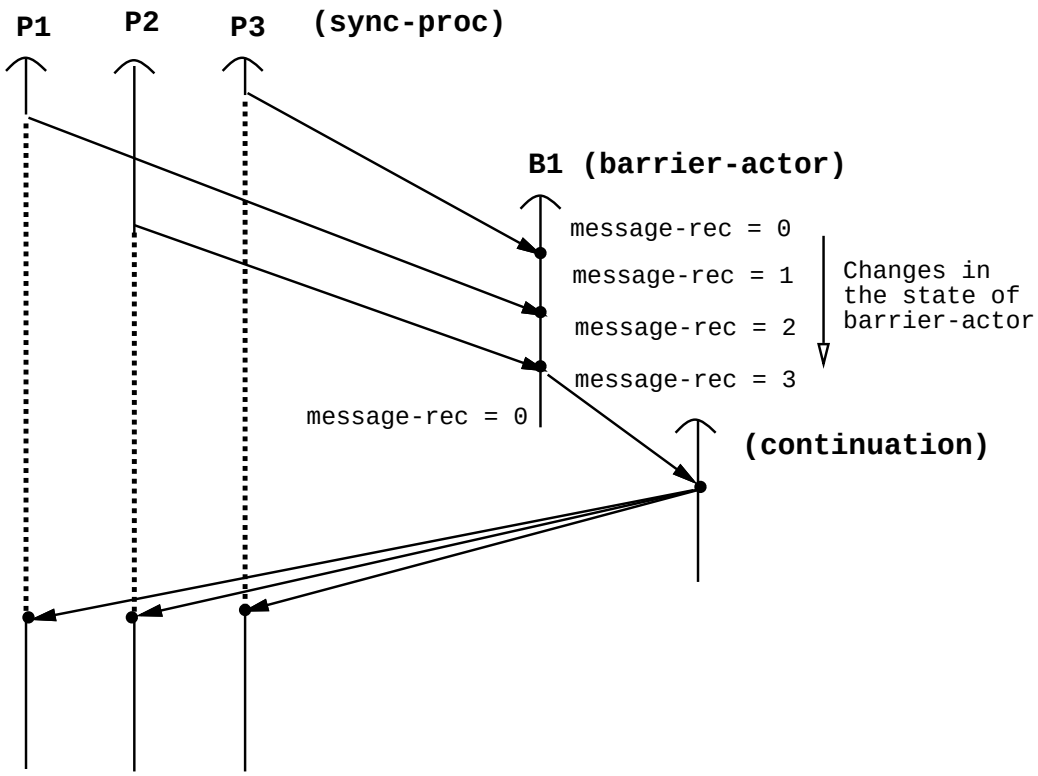


Figure 1: Implementing Barrier Synchronization using Asynchronous Message-passing in Actors

where k is a constant. A better implementation partitions the array alternately in the vertical and horizontal direction to increase locality. This improves scalability as reflected by the corresponding isoefficiency function:

$$W = \Theta(2^{\sqrt{kp}} \times \sqrt{p}).$$

Given a scalability measure such as the isoefficiency function and the input size, the system can decide the number of processors to be used to solve the problem. In practice, the system should consider the overall load and also the priority of the computation (explained below) for deciding the number of processors.

We discuss another algorithm, the Cholesky Decomposition to illustrate how the number of processors affects the efficiency of an algorithm. Figure 3 gives a comparison of two different implementations of the Cholesky Decomposition algorithm on matrices of size 32×32 and 64×64. The functional implementation synchronizes after every iteration of the algorithm whereas the concurrent implementation lets several iteration of the algorithm run concurrently. It can be seen that the speedup obtained by parallel execution of the algorithm initially increases as the number of processors is increased, but starts starts decreasing as the communication overhead overcomes the increase in speedup obtained by the added processors. Thus given a proper measure of the scalability of the Cholesky Decomposition algorithm, the system could decide the optimal number of processors to be used for a given size of the input matrix.

- *Scheduling Computations:* On a network with several users, a priority may have to be assigned to a computation. Thus, a computation with high priority may be run using a large number of processors even though the efficiency of the execution may be low because of scalability criteria. Also, if several other computations are dependent on one computation system may decide to increase its priority since slow execution of this computation will delay the execution of several other computations. In such a case the computation may be executed using a large number of processors so that the overall time taken is less even though the efficiency is low.

Such metrics help the system decide issues such as the parallelism available in a computation, the communication overhead associated with the parallel execution etc. The system should have a function mapping the various values of the above indices to different

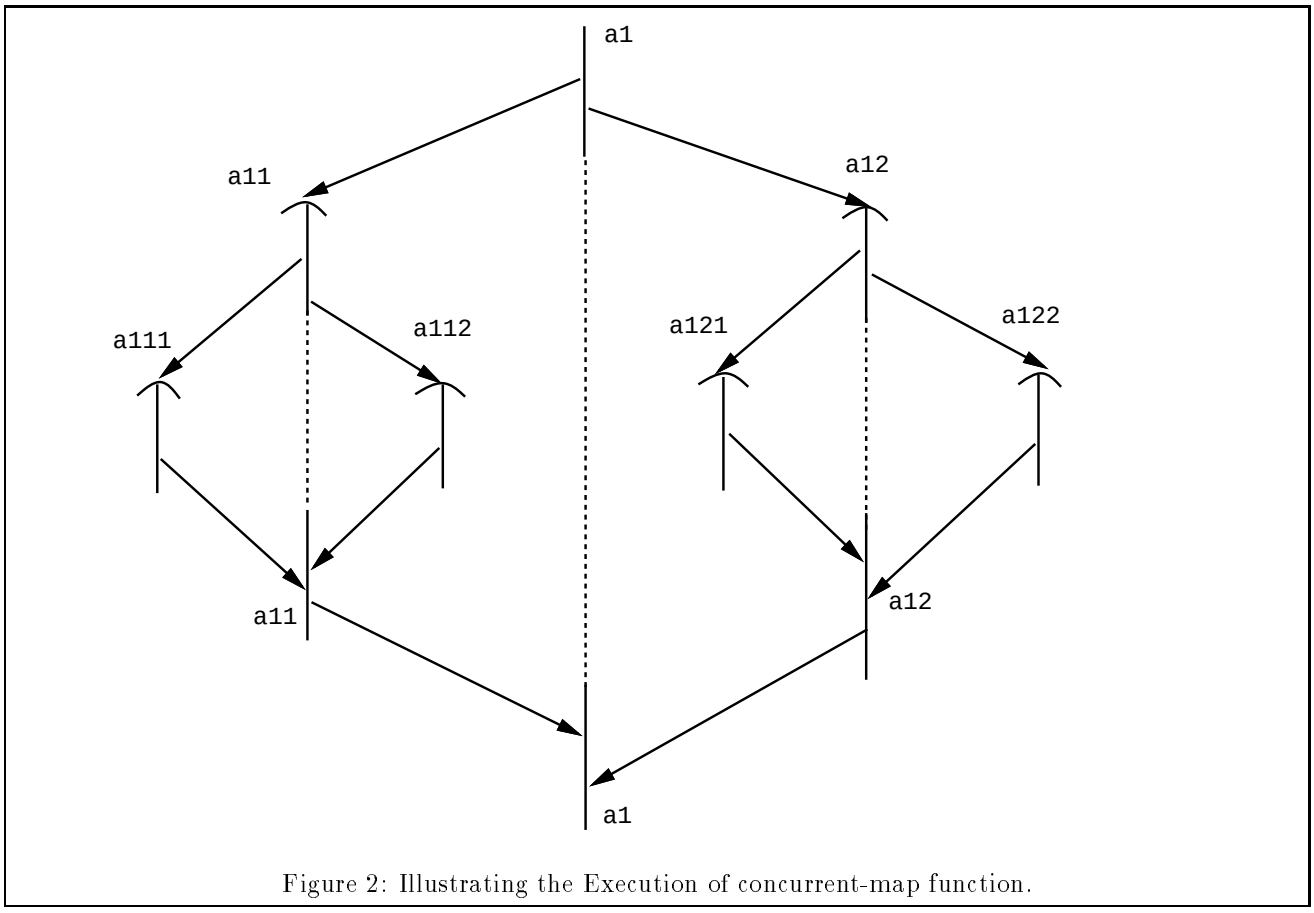


Figure 2: Illustrating the Execution of concurrent-map function.

architectures. This function may be implemented as a table or as a heuristic based algorithm. For example, for an application such as low-level image processing, the concurrency index is usually very high, the grain size is small and the communication/computation ratio is low. Given the values of such metrics, the system should choose the Connection Machine as one of the suitable architecture for such an application.

For an application that can be divided into several sub-problems, each of which has very different characteristics, the above indices should be provided for all such sub-problems. Thus, the system has to decide where each portion of the code has to be executed and the number of processors allocated for each sub-problem. In certain special cases where the code is very simple, it may be possible to decide the values of such indices statically at compile time.

3.1 HIGH-LEVEL RESOURCE CONTROL

A number of problems involve speculative parallelism. For example, there may be several methods for solving a problem. In such cases, speed-up may be obtained by computing multiple methods concurrently. One can characterize such problems as search problems. In fact, in some search problems, superlinear speed-up may be obtained by coordinating multiple processes.[2] In search problems, the computational space may grow exponentially; an intelligent search scheme uses partial results to control the speculative activity. We discuss a specific example, namely *branch and bound search*, to illustrate how high level resource control may be used to optimize search using appropriate heuristics. The high-level resource control provides an abstraction which may be used to improve the efficiency of execution in a heterogeneous network.

A typical *branch and bound* search consists of computing an upper (or lower) bound on the value of the optimal solution at each node and expanding the search in the neighborhoods of nodes with the best bounding values. Implementing such a search requires a number of distinct interlinked computations to be carried out. The required computations can be summarized as follows:

node calculation: local computation on a node graph to compute neighbors for further expansion; and,

resource allocation: how much resource should be allocated – no resources indicates the node is not to be expanded at the present time. The resources

[2] Of course, in such cases the original sequential algorithms are sub-optimal and could be replaced by a time-sharing system.

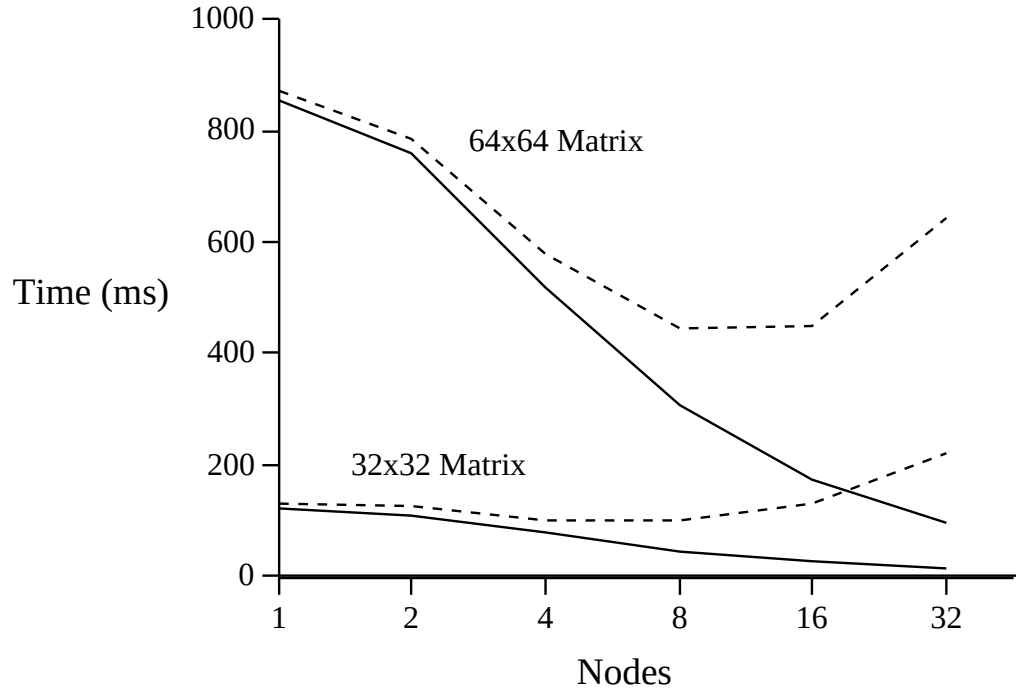


Figure 3: Performance of Cholesky Decomposition Algorithm on Hypercube as a Function of the Number of Processors. *Dashed lines represent the functional algorithm. Solid lines represent the more concurrent implementation.*

provided will determine the extent of the expansion. Note the extent of expansion around a node requires global as well as local information.

Separating the above computations into different modules provides a separation of design concerns: for example, high-level resource allocation policies may be modified independently of logic of the node expansions. Heuristics will then be used for partitioning the problem on a particular architecture and computing costs to be assessed.

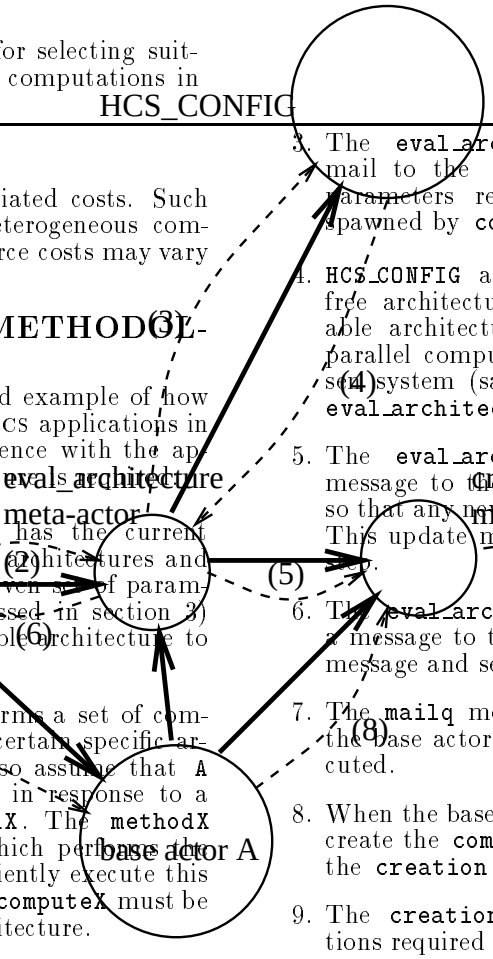
We use a sponsorship mechanism which requires all activity to be charged to sponsors. A computation activating a number of new sub-computations is required to provide resources for those activities. Furthermore, it may specify a sponsor which holds onto more resources and allocates them dynamically based on intermediate results – thus affecting the course of the search as it proceeds. Sponsors may be dynamically spawned, and resource control subdivided, so that they do not themselves become a bottleneck.

Consider the following code. In order to determine how much resource to give to a node expansion, the sponsor below uses a procedure called `sponsorship-algorithm` which is parameterized by `current-resources` available to the sponsor. Thus the sponsor may drive or throttle a particular computational path. Assume that `node` appropriately responds to `expand` messages (or equivalently that `expand` is a procedure which may be applied to `node`). We use the `with` annotation to provide a hook for the system to directly access physical resources in the underlying architecture.

```
  (procedure node-sponsor
   (mutable [node current-resources
sponsorship-algorithm]
     [add-resources
       (let [[new-resources (+
current-resources add-resources)]]
         (become node-sponsor [node
new-resources sponsorship-algorithm]))]
     [expand-node
       (letrec [[descendant-nodes (expand
node (with current-resources))]
                [sponsorship-level
(sponsorship-algorithm current-resources)]
                [new-sponsors (map
sponsorship-level descendant-nodes)]
                [new-resources (apply + (map
current-resources new-sponsors))]]
         (become node-sponsor [node
new-resources sponsorship-algorithm]))]
     [change-algorithm
          ....]
     [request-resources
          ....]))
```

The decision about *whether* to perform a subcomputation is thus specified independently of *how* the computation itself is to be carried out. Furthermore, this sponsorship algorithm leaves open the decision on how to utilize the physical resources, such as memory

Figure 4: Metalevel architecture for selecting suitable architecture for execution of computations in an application.



or processing power, or their associated costs. Such flexibility is critical in building heterogeneous computing systems where relative resource costs may vary with different architectures.

4 A PROGRAMMING METHODOLOGY FOR HCS

In this section we give a detailed example of how reflection can be used to program HCS applications in a modular fashion without interference with the application code. The following structure is assumed:

1. An actor `HCS_CONFIG` which has the current knowledge about the available architectures and resources on the HCS. For a given set of parameters of a computation (discussed in section 3) it can evaluate the best available architecture to perform the computation.

2. Assume that an actor `A` performs a set of computations which are suited to certain specific architectural characteristics. Also assume that `A` starts this set of computation in response to a message that invokes `methodX`. The `methodX` creates an actor `computeX` which performs the required computation. To efficiently execute this set of computations, the actor `computeX` must be created at an appropriate architecture.

3. To enable `A` to evaluate a suitable architecture for executing `computeX`, the following metelevel actors of `A` need top be reified.

   (a) A `mailq` metelevel actor that traps all messages that invoke `methodX`.

   (b) A `eval_architecture` metalevel actor that keeps a list of all parameters related to the parallel computation performed by `computeX` as discussed in section 3.

   (c) A `creation` metalevel actor which contains the information regarding target architectures or processors where new actors are created by `A`. Thus, every `new` command defined in the method definitions of `A` sends a message to the `creation` meta-actor which decides the placement of the new actor and then creates it appropriately.

The steps performed by the above metalevel architecture are as shown. The step numbers mentioned here correspond to the numbers shown in Figure 4.

1. A message `mx` that invokes method `methodX` is received by the `mailq` meta-actor of `A`.

2. This message is trapped at the `mailq` actor and the rest of the messages are forwarded to the base actor `A`. `mailq` sends a message to the `eval_architecture` meta-actor to compute the best available architecture for placing `computeX`.

3. The `eval_architecture` meta-actor sends a mail to the `HCS_CONFIG` actor, giving it the parameters related to the parallel execution spawned by `computeX`.

4. `HCS_CONFIG` actor has the list of the currently free architecture and evaluates the best available architecture for the given parameters of parallel computation. The address of the chosen system (say `M`) is communicated back to `eval_architecture` meta-actor.

5. The `eval_architecture` meta-actor sends a message to the `creation` actor and updates it so that any new `computeX` actor is created on `M`. This update must be completed before the next step.

6. The `eval_architecture` meta-actor then sends a message to the `mailq` actor to release the `mx` message and send it to the base actor `A`.

7. The `mailq` meta-actor sends the message `mx` to the base actor `A`. The code for `methodX` is executed.

8. When the base actor `A` executes the command to create the `computeX` actor, it sends a message to the `creation` meta-actor.

9. The `creation` actor takes the appropriate actions required to create `computeX`.

It can be seen that if `methodX` itself performs the required computation (instead of creating the new actor `computeX`), it has to be migrated to architecture `M` before the message `mx` is released by `mailq` and sent to the base actor. It is the users responsibility to identify the methods tha start the main parallel computations of the application that require execution on specific architectures and reify the appropriate metelevel actors as shown above.

Having decided on what architecture to use for execution of a certain part of the code one has to make several low-level decisions on resource management while executing the required code. These decisions may be architecture sensitive and appropriate changes have to be made to the code depending on the target architecture. The section below discusses one of the important resource management issues namely the placement of actors on different processors.

5 MODULAR SPECIFICATION OF ACTOR PLACEMENT

An important criteria for efficiently executing parallel programs on a multiprocessor is the placement of data on processors. This affects the load balancing of computation and also the communication overhead between different processors. Placement of data can be treated completely seperately from the computation thereby getting different efficiency values for different placements. For simple parallel computations one can specify the placement of data elements and tasks statically. Complicated placement strategies like dynamic

placement of tasks and data elements, require an algorithmic description for specification.

Efforts have been made to specify placement using annotations in the program. These anotations occur in the code whereever new data elements or new tasks are created and specify the processor on which the tasks/data elements should be created. As a result the code specifying the placement is mixed with the code specifying the application. If a new placement strategy is used, changes may have to be made at several places in the code. In an HCS the placement strategies may be changed depending on the architecture used to solve a particular part of the application. In a programming environment which requires annotations for specifying placement a new version of the code is required whenever the placement strategy is changed. Reflection can be used specify the placement issues in a modular fashion and the placement strategies can be changed easily at run time.

The example shown below illustrates how placement strategies can be architecture dependent.

5.1 EXAMPLE

We consider the example of Cholesky decomposition discussed in [4]. The optimal placement strategy for such a problem depends on several factors such as the communication overhead for point-to-point messages on the architecture, the overhead required for a full broadcast of a message, the size of the given matrix, the number of processors available etc. In the Chlolesky decomposition algorithm (and also the Gaussian Elimination algorithm) the active part of the matrix (which is being used as input for subsequent computation and the part which is being updated) during the i^{th} iteration is composed of rows from i to n.

Two example placement strategies are described below:

1. Static placement in which row i is given to the processor i div P, P being the total number of processors. This strategy causes unequal load balancing since the processors containing rows with indices less than j are idle after the $(j-1)^{th}$ iteration is over.

2. The row i is given to the processor $i \bmod P$. This strategy causes almost equal load balancing throughout the execution of the algorithm but involves a full broadcast of the i^{th} row for carrying out the i^{th} iteration.

It seems that strategy 2 causes better load balance than strategy 1 and may result in faster execution. The strategies 1 and 2 were implemented on the intel's iPSC/2 hypercube and for several executions the performance of strategy 1 was observed to be better than strategy 2.

The above performance is observed because strategy 2 causes several broadcast operations which strategy 1 doesn't. If an architecture with extra hardware for speeding up broadcast operation is available, the strategy 2 may perfom better that 1. This example illustrates how the placement decision may be architecture sensitive. Currently there are no known methods to analyse a given algorithm and decide the placement issues automatically, the user has to provide the required information. Reflection can be used to provide modular specification of placement issues.

nodes	32×32	64×64	128×128	256×256
1	101	768	6068	48203
2	91	691	5416	42919
4	69	477	3600	28734
8	55	305	2098	16265
16	28	293	1371	8940
32	23	273	1097	6033

Table 1: Results from an implementation of the Cholesky algorithm on an Intel iPSC/2 with an equal number of matrix rows per processor (strategy 1). *Times are in milliseconds.*

nodes	32×32	64×64	128×128	256×256
1	145	1032	7848	61156
2	125	619	4167	31202
4	84	386	2348	16464
8	185	417	1627	9420
16	276	546	1551	6597
32	*none*	851	2010	6585

Table 2: Results from an implementation of the Cholesky algorithm on an Intel iPSC/2 with a shuffled row distribution (strategy 2). *Times are in milliseconds.*

For static placement strategies the modification of the `creation` actor is appropriate for deciding the placement of new actors being created. For implementing dynamic placement strategies a `placement` meta-actor is added. This meta-actor receives messages and evaluates certain functions to compute the new placement of the actor. If the new placement is different from its current placement, the actor is migrated to the new processor. The messages that trigger the placement can be generated by an additional level of `mailq` meta-actor that traps some specific incoming messages. For example, for the Cholesky Decomposition example, the messages requesting initiation of the next iteration can be trapped. Based on the iteration number, at certain intervals the placement of every actor can be re-evaluated, causing migration if required.

6 CONCLUSIONS

A number of areas of research need to be developed to enable effective use of HCS. Specifically, we have identified three such areas. First, further research is needed in determining how to best combine scalability metrics and architectural characteristics for efficient execution. Second, a fully reflective system ar-

chitecture that allows a high-level representation of resources needs to be developed. Finally, a library of concurrency abstractions suitable for using heterogeneous processing networks needs to be implemented.

References

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

[2] G. Agha. Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE '89)*, pages 1–19. Espirit, Springer-Verlag, 1989. LNCS 366.

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

[4] G. Agha, C. Houck, and R. Panwar. Distributed execution of actor systems. In *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, 1991.

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

[6] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*, 1991. LNCS 480.

[7] V. Chaudhary and J. K. Aggarwal. Parallelism in computer vision: A review. In V. Kumar, P. S. Gopalakrishnan, and L. N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 271–309. Springer-Verlag, 1990.

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

[9] W. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Press, 1986.

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

[11] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[12] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[13] Y. Sami and G. Vidal-Naquet. Formalisation of the behaviour of actors by colored petri nets and some applications. In *Proceedings of Parallel Architectures and Languages Europe, (PARLE '91)*, 1991.

[14] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *Proceedings of the International Parallel Processing Symposium*, pages 92–101. IEEE, 1991.