Runtime Systems for Fine-Grain Multicomputers

Thesis by

Nanette Jackson Boden

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California USA

1993

(Submitted January 20, 1993)

Caltech-CS-TR-92-10

Acknowledgments

I would first like to thank the members of my thesis committee, Jan van de Snepscheut, Steve Taylor, Herb Keller, Eric Van de Velde, and of course, Chuck Seitz for their comments on this work and for their support throughout my years in grad school.

I thank AT&T for their supporting my research with an AT&T Doctoral Fellowship. In addition to giving AT&T my long-distance dollars with a glad heart, I will continue to lecture every MCI or Sprint solicitor about how AT&T's technological vision sets them apart from the rest.

My deepest thanks go to Chuck. He guided my development as a researcher without controlling it. He showed me his vision of computer science, while helping me develop my own vision. He took a chance on an unknown and gave this Alabama girl the opportunity to get a Ph.D. from Caltech – for which I shall eternally be grateful. He has been the very best advisor I could have had.

Thanks also go: To Arlene Desjardins and the members of Chuck's research group who taught me alot about graduate school, research and life in general: Bill Athas, Wen-king Su, John Ngai, Craig Steele, Mike Pertel. Jakov Seizovic gets a separate sentence of thanks for C+-, for the good work we've done together, and for his friendship.

To my fellow grad students, especially Pieter Hazewindus, for their friendship. To my long-time office-mate Andy Fyfe for his friendship, and being the Nan-friendly Unix guru that he is.

To the teachers who made the difference along the way, Elizabeth Brett, who showed me through her teaching and through her example how civilized people should behave, and to Dr. Cathy Randall, whose enthusiasm and grace have made her one of my most persistent role models.

To my buddies, Steve Burns, Michael Emerling and Ruth Ballenger, Mass and Ruth Sivilotti. I'd always heard about the friendships that last a lifetime – now I know how much fun they can be.

To my husband, Andy (The Bode) Boden. I thank him for technical help when I needed it, and for being my Best Buddy all the time.

To Dad, and especially Mom, for all the late-night phone calls, the pep-talks, the encouraging letters, for telling everybody in town what I was doing, for instilling their hyperactive work ethic in me, for their unwavering support. I could never have done any of this without you.

The research described in this thesis was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) Submicron Sustems Architecture Project.

To Mike and Marinan, Andy and Tyler

Abstract

During the past decade, our research group has been engaged in experiments in the architecture and programming of multicomputers. This research has progressed steadily toward the ideal of *small granularity,* both of the computing nodes within a multicomputer, and of the execution units within concurrent programs. The context for the runtime-system and program-behavior experiments reported in this thesis are: (1) the reactive-process, message-passing computational model, (2) C+−, a C++-based, concurrent-programming notation, and (3) the Mosaic C, an experimental, fine-grain multicomputer.

We present first a long-sought solution to the formulation of an unbounded queue of elements within the reactive-process model. This result is applied to allow messages to be received selectively using purely reactive semantics.

The primary contributions of this thesis are distributed algorithms and a design method for runtime systems for fine-grain multicomputers. To evaluate the algorithms and design, a prototype runtime system called MADRE was developed, C+− programs whose behaviors are typical of a variety of applications were written, these programs were executed on the Mosaic C under MADRE, and the program behavior was instrumented.

In addition to conventional operating- and runtime-system functions such as local memory management and quiescence detection, MADRE automatically manages user-process placement and naming. MADRE can also be configured to include capabilities for distributing resource demands across the nodes of the multicomputer. Buffered messages can be exported from congested nodes so that incoming messages can continue to be received. The code of user programs can be distributed across the ensemble, and accessed automatically. Each of these capabilities depends upon the formulation of selective receive demonstrated in the solution to the unbounded queue.

Our experiments evaluate various automatic process-placement strategies. We show that one algorithm, called k-biased placement, distributes loads nearly as well as random placement, while providing a tunable degree of locality between parent and child processes. Other experiments demonstrate that the message-exportation capability is crucial to fine-grain multicomputers; unless messages can be exported, computations fail due to receive-queue overflow when only a fraction of the multicomputer's memory resources are being used.

Contents

List of Figures

List of Tables

List of Programs

1 Introduction

1.1 Rules of the Game

Imagine a computation expressed as a collection of small processes that communicate by messages. The cost of a message-passing operation is roughly equivalent to the time required for a procedure call. FIFO-buffered, asynchronous channels comprise the message network. The order of messages sent between pairs of processes is preserved — messages arrive, after an arbitrary travel time, at the destination process in the order in which they were sent from the sender process. However, if two messages are sent from the sender process via different intermediate processes, the arrival order at the final destination process is non-deterministic.

Each process is identified by a unique *reference value* that acts as its address for message passing. A process is defined by a body of code to be executed and a small set of private variables. New processes can be created dynamically at a cost that is only slightly more than the cost of sending a message. The parent process initializes some or all of the private variables of the new process. As part of creating a new process, the reference value of the new process is returned to the parent. A process may obtain a reference value either as an initialized variable, by having created the process, or by receiving the value in a message.

Since each process contains little state, it relies on passing messages to obtain new tasks and new data. Just as in the Actor model of computation [1], a process executes code for a bounded period of time in response to the arrival of a message. Based on the state of the process and the contents of the message, this code may create new processes, send messages, and modify its own state. These semantics were heavily influenced by the Actor model of computation [1].

When receiving a message, the process *must* accept the message at the front of the

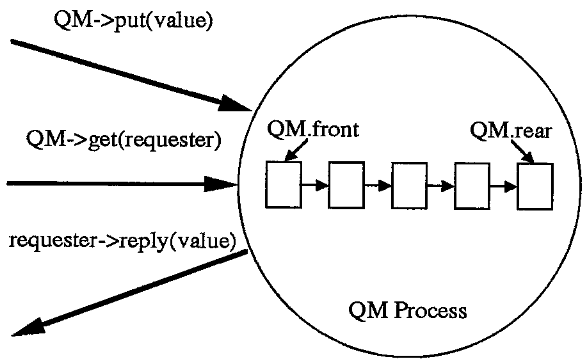


Figure 1.1: Single-Process Unbounded-Length Queue. Producer/consumer processes interface with the queue-manager process (QM) using put, get and reply messages.

queue (*ie,* the message that has been in the queue for the longest time). No hidden facilities for buffering unwanted messages are provided; each message must be processed in the order received. This *reactive* handling of messages [39, 4] is a significant departure from traditional protocols used for process interaction. *Selective-receive* capabilities, the ability to receive messages based on their contents and/or the state of the process, must be implemented using purely reactive semantics.

These groundrules outline a model of computation that has developed along with the evolution of the multicomputer architecture. The factors that motivate this model are discussed in the literature [30, 39] and in subsequent sections of this chapter.

1.2 The Unbounded Queue Problem

1.2.1 A Single-Process Implementation

Figure 1.1 and Program 1.1 illustrate an implementation of a single-process, unbounded-length queue using the groundrules described in the previous section. The specification for this queue is:

- a put operation appends the argument value to the queue rear,
- a get operation eventually returns the value at the queue front in a reply message,

```
class element
{ int                     value;
  element*                next;
  public:
                          element(int V)                  { value = V; next = NIL;}
        void              set_next(element *N)                     { next = N;}
        int               get_value()                         { return value; }
        element           *get_next()                          { return next; }
};

processdef QM
{ element*                front;
  element*                rear;
  public:
                          QM()                { front = rear = (element *) NIL;}
        atomic void       put(int);
        atomic void       get(USER *);
};

void
QM::put(int value)
{ element* new_rear = new element(value);

  if (rear == (element *) NIL)
        front = new_rear;
  else
        rear→set_next(new_rear);
  rear = new_rear;
}

void
QM::get(USER *requester)
{ if (front == NIL)
        ERROR_UNDERFLOW();

  requester→reply(front→get_value());
  front = front→get_next();
}
```

Program 1.1: C+− Program for a Single-Process, Unbounded-Length Queue.

- through this interface, multiple producer and/or consumer processes should be able to use the queue as a FIFO data structure.

Program 1.1 is written in C+-, a programming notation based on C++ that has been developed in our research group at Caltech. The mission of this programming notation is to apply the essentials of object-oriented programming and modern compilation techniques to multicomputer programming. As most of the programs in this thesis are written in C+-, the reader should already be familiar with C++ (or study a good C++ programming reference such as [37]). The primary differences between C+- and C++ are listed below.

- C+- does not permit global variables.

- C+- adds a process abstraction (indicated by the keyword **processdef**) that is an extension of the C++ class abstraction. Processes are created using the **new** operator; the return value of the **new** operation is the reference value of the new process.

- Process member functions can be annotated as **atomic**, indicating that invocation of these functions is equivalent to sending a message to the process, which then executes the function on the given arguments. By invoking an atomic function, a process can trigger delayed and independent computation, yielding potential concurrency. Atomic functions represent the fundamental semantic difference between C++ and C+-.

- The atomic functions of a process can be enabled/disabled by the process so that messages can be selectively received.

Chapter 3 includes a description of the C+- notation, along with example application programs.

In Program 1.1, a pool of producer-consumer USER processes send **put** and **get** messages to the queue-manager (QM) process. The QM process maintains an internal linked list of **elements** to queue incoming **values**. Upon receiving a **get** request, the QM process removes the value at the front of the internal queue and returns it to the requesting USER process via the **reply** member function. It then updates the **front** of the queue to point to the next **element**.

This implementation satisfies our specification of a queue since the FIFO channels and the operation of the queue-manager process ensure that values that were **put** into the queue are received *in order* in response to **get** operations. This implementation does not, however, satisfy the programming groundrules described in Section 1.1. In that model growth of the computation is expressed *using process creation*, whereas Program 1.1 relies on the growth of internal process state. A better solution to the unbounded queue problem would use distributed processes to implement the queue of values.

1.2.2 A Solution Using Distributed Processes

The question of how to use distributed processes to implement an unbounded queue has been studied for roughly a decade [21, 4, 8]. Most of the problem "solutions" were either restricted to simplified forms of the unbounded queue problem or were quite wasteful of space and/or work. The development of an acceptable solution within our computational framework was a key starting point for the research in this thesis. In this section, we present this solution to the distributed queue problem that satisfies all the constraints outlined above and is efficient in terms of both space and work. In subsequent sections, we describe and analyze some of the earlier, unacceptable, solutions, and expose some of the finer points about this new solution.

Since an individual process may contain little state and there may be an arbitrary number of elements in the queue, the data structure for the queue elements will contain one element per process. This choice of one element of the queue per process is for simplicity of exposition. In the general case, the structure of the program is the same if a queue process represents a single element of the queue or a sequence of elements. Figure 1.2 illustrates the reference connectivity of the queue-manager (QM) process and a set of queue-element (E) processes.

The difficulty with simply extending the solution in Program 1.1 to a distributed

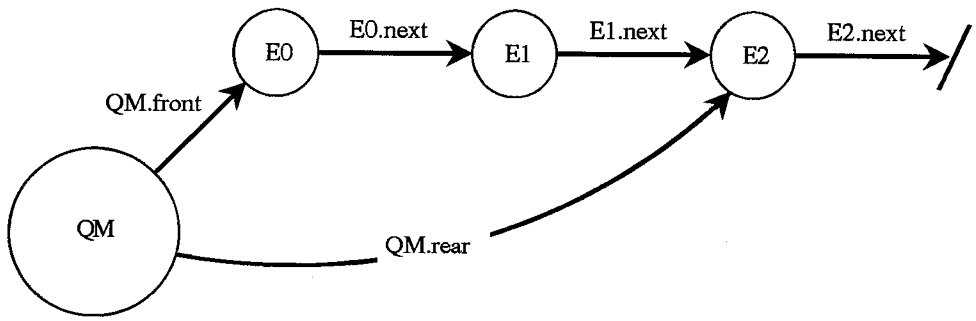


Figure 1.2: Queue of Processes Used to Implement an Unbounded-Length Queue. (Circles represent processes, heavy lines indicate reference connections).

implementation appears in the **get** operation. When a **get** message arrives at the queue manager, it should relay the **get** message to the queue element at the front of the queue, which will eventually reply with the desired value and the reference to the new queue front. However, what happens if, before the reply message is received from the queue, another **get** message arrives at the queue manager? Since the old queue-front reference is no longer valid and the new one has not yet arrived, the queue manager is effectively cut off from the queue, meaning that *the queue manager must somehow buffer the unwanted* **get** *messages until the new queue front reference arrives.* At first, this dilemma may seem nothing more than an artifact of the algorithms and approach used to attack this problem. However, further consideration shows that this problem is more fundamental: what we are attempting is *to receive selectively a particular message, while buffering all unwanted messages.* Recall that this type of buffering is expressly *not* provided in our model.

An obvious solution to this problem is to constrain the queue's environment so that the problem does not arise. In [4], Athas defines the environment as a single-consumer process that does not issue a **get** message until the reply from the previous **get** message has arrived. In this case, the queue manager is guaranteed to be reconnected to the queue before the next **get** message arrives. There are two problems with this approach. First, it is a solution to a different problem from the one initially posed, a general multiple-consumer queue. Second,

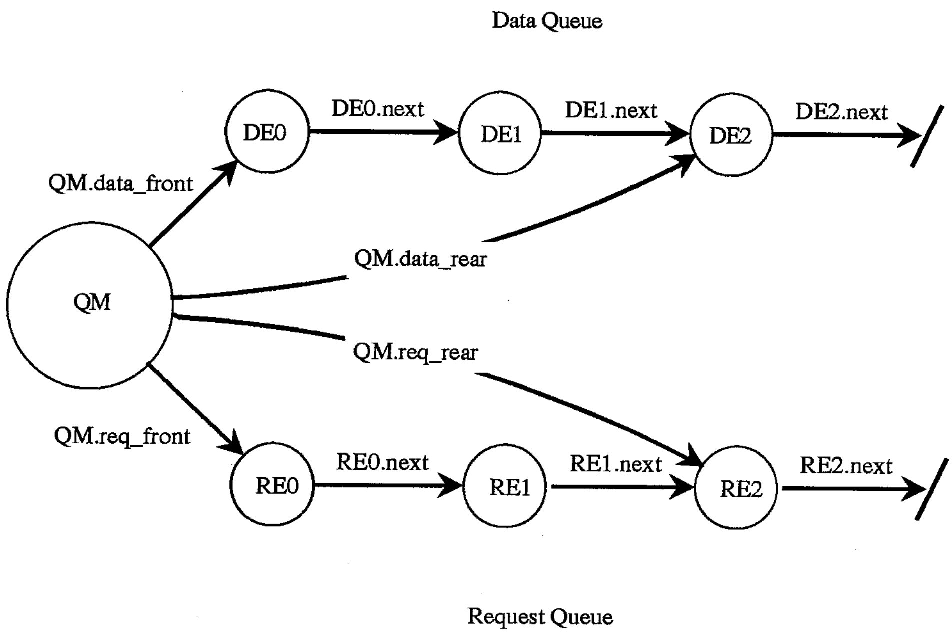


Figure 1.3: Process Structure of a Distributed Unbounded-Length Queue. The queue-manager QM process maintains two queues — the data queue, to store values queued by producer processes, and, the request queue, to store requests for values from consumer processes.

even in the restricted, single-consumer solution, the behavior of the consumer may exhibit the same dilemma as outlined above. For example, if the consumer's program causes it to respond to messages from other processes by sending **get** messages to the queue, then it must buffer those incoming messages until the reply to the pending **get** message is received. Consequently, the consumer must selectively receive a **get** message reply while buffering the unwanted messages from other processes, thus encountering the same difficulty as the queue manager in the general queue implementation.

A solution for implementing the selective receive for messages returning from the queue containing data is to use another queue to buffer the **get** request messages that cannot be processed immediately. Figure 1.3 and Programs 1.2 – 1.4 illustrate the use of two such queues.

```
processdef DE
{ int           value;
  DE*           Next;
  QM*           qm;
  public:
                DE(int v,DE*n,QM*q)      { value = v; Next = n; qm = q; }
    atomic void next(DE*n)                                  { Next = n;}
    atomic void get(USER*,int);
};

processdef RE
{ USER*         requester;
  RE*           Next;
  QM*           qm;
  public:
                RE(USER*u,RE*n,QM*q)  { requester = u; Next = n; qm = q; }
    atomic void next(RE*n)                                  { Next = n;}
    atomic void get(int);
};

processdef QM
{ DE*           data_front,*data_rear;
  RE*           req_front, *req_rear;
  public:
                QM()                     { data_front = data_rear = NIL;
                                             req_front = req_rear = NIL; }
    atomic void put(int);
    atomic void get(USER*);
    atomic void data_reply(int,USER*,DE*,int);
    atomic void req_reply(USER*,RE*,int);
           void data_queue_get(USER*u)
                {  data_front→get(u,data_front == data_rear);
                   if (data_front == data_rear)
                      data_rear = NIL;
                   data_front = (DE*) UNDEF;
                }
};
```

Program 1.2: C+− Program for a Distributed Unbounded-Length Queue (a).

```
void
DE::get(USER *u,int b)
{       qm→data_reply(value,u,Next,b);
        delete this;
}

void
RE::get(int b)
{       qm→req_reply(requester,Next,b);
        delete this;
}

void
QM::put(int v)
{       DE *qe = new DE(v,(DE *) NIL,this);
        if (data_rear ≠ (DE *) NIL)
          data_rear→next(qe);
        else
          data_front = qe;
        data_rear = qe;
}

void
QM::get(USER *u)
{       if (data_front == (DE *) NIL)
                ERROR_UNDERFLOW();
        else
          if ((data_front == UNDEF) || (req_front ≠ (RE *) NIL))
          {
                RE *re = new RE(u,(RE *) NIL,this);
                if (req_rear ≠ (RE *) NIL)
                  req_rear→next(re);
                else
                  req_front = re;
                req_rear = re;
          }
          else
                data_queue_get(u);
}
```

Program 1.3: C+− Program for Distributed Unbounded-Length Queue (b).

```
void
QM::data_reply(int v,USER *u,DE *qe,int singleton)
{         u→reply(v);
          if (!singleton)
                    data_front = qe;
          else
            if (data_rear == (DE *) NIL)
                    data_front = NIL;
          if (req_front ≠ (RE *) NIL)
            if (data_front == (DE *) NIL)
                    ERROR_UNDERFLOW();
            else
            { req_front→get(req_front == req_rear);
                    if (req_front == req_rear)
                      req_rear = NIL;
                    req_front = (RE *) UNDEF;
            }
}

void
QM::req_reply(USER *u,RE *re,int singleton)
{         data_queue_get(u);
          if (!singleton)
            req_front = re;
          else
            if (req_rear == (RE *) NIL)
              req_front = NIL;
}
```

Program 1.4: C+− Program for Distributed Unbounded-Length Queue (c).

The basic operation of the queue-manager process is as follows. Values in **put** messages received by the queue manager are appended to the data queue, while **get** messages received are relayed to the the data queue only if no **get** message is currently being processed and no previous **get** requests are buffered. If either of these conditions is false, the **requester** value in the received **get** message is appended to the rear of the request queue.

The same put mechanism is used for each of these two queues. The queue manager creates a new queue-element process (a DE process for the data queue or a RE process for the request queue) to hold the argument (a **value** or a **requester** reference). It then appends the process to the queue by invoking the **next** function of the queue-rear process. Because the **next** message is sent *directly* from the queue-manager process to the queue-rear process, message-order preservation ensures that the **next** message will arrive at that process before a **get** message. In addition, this put mechanism can be used even when the queue manager is effectively disconnected from the front of a queue. In those cases, the queue manager still possesses the reference to the *rear* of the queue, the only reference needed to append processes. Thus, **put** operations can always be processed *immediately.*

This point is crucial. The message-order-preservation property dictates that any request messages that have been buffered must be processed before any incoming request messages. Thus, when the **data_reply** message is received from the data queue, any **get** messages in the request queue must be retrieved and sent *in the order in which they were originally received* to the data queue. The queue manager must, accordingly, issue a **get** operation to the request queue. Just as the **get** operation on the data queue disconnects the queue manager from the data queue, this **get** operation disconnects the queue manager from the request queue.

Before the **req_reply** is received, additional **get** request messages may arrive and need to be buffered. Using the **put** operation, the **requester** values in these messages can be appended to the request queue *immediately.* When the **req_reply** message is received, the queue manager sends a **get** message with the enclosed **requester** reference to the data queue. Thus, the queue manager never issues a **get** message to the request queue until the reply to the previous **get** message to the request queue has been received and processed.

Requiring an additional unbounded queue (here, the request queue) to implement an unbounded queue (the data queue) might appear to be a recursive requirement. Wouldn't one need yet another unbounded queue to implement the second queue, and so on? The error in this logic stems from classifying the request queue and the data queue as being the same type of queue. They use the same protocols and mechanisms for **put** and **get** operations, but their environments differ in an important way. The data queue is an unrestricted multiple-producer, multiple-consumer queue: an unbounded number of **put** and **get** operations may be sent to the queue manager by producer and consumer processes. The request queue, however, is sent **put** and **get** messages only by the queue manager. More importantly, the queue manager does not send a new **get** message to the request queue until the reply from the previous **get** message is received. Thus, the request queue is an instance of Athas's constrained queue solution mentioned above, and described in [4]. Since the multiple-producer, multiple-consumer queue implementation relies only on a known solution to the simpler problem, there is no recursive requirement.

The solution to the general unbounded queue problem that has been presented is an improvement over earlier solutions and approaches (see next section). However, the reader may have noticed that the **put** mechanism, which is pivotal to our solution, relies heavily on one of our programming groundrules. The new queue-rear process is created by the queue manager, which then sends its reference value immediately to the old queue-rear process. Since reference values are returned immediately to the parent process in our model, two-way communication between the parent process and the child process is expected. What if other messages arrive at the parent process before the reference value of the child process is received? Are we not attempting to receive the reply selectively from the child process? If the process-creation mechanism itself requires a selective receive, our solution to the general queue problem is recursive, and therefore unacceptable (recursion requires the growth of internal data structures). A reactive implementation of the process-creation mechanism is, indeed, crucial to our solution, but we shall see in Chapter 5 that it can be implemented efficiently using purely reactive semantics.

1.2.3 Previous Work on the Unbounded-Queue Problem

The problem of implementing an unbounded queue using only reactive semantics has been misunderstood for about a decade. This problem seemed to frame the conflict between reactive semantics, whose properties are ideally suited for multicomputers (section 1.3.3), and a selective receive, a programming mechanism so powerful that no "real" system would be complete without it. Fortunately, armed with a solution, we see now that there is no inherent conflict — selective receive can be implemented efficiently using purely reactive semantics.

Members of our research group had searched so long for algorithms to solve this problem that some believed there was in fact no solution that was both efficient and general. However, because a selective receive is such a convenient programming mechanism, our software systems included it as a primitive operation. In [21], Lang added a SELECT construct to Simula so that the programmer could control the order of messages to be processed. In the original Cosmic Kernel system [28], programmers could embed type information in outgoing messages and specify the desired type of messages to be received. The Reactive Kernel system [33] provides message discretion via library functions that are layered on a kernel that handles messages reactively. In [8], we proposed that a *remote function* construct be added to the reactive programming language Cantor [4] (section 1.3.2) to provide the programmer with a selective receive. This construct, in turn, relied on a system-level mechanism called a *custom function* to buffer unwanted messages.

During the search for a solution to the general unbounded queue problem, we developed a family of queue implementations that satisfy the criteria of queue operation, yet are wasteful of space and/or work. Such queues typically consist of a set of processes that never decreases in size; new processes are created as needed but none are ever deleted. Each message to and from[1] the data queue traverses much of the list of processes. Since the set

[1] Just as for the queue-element processes described in the previous section, all messages returning values from the queue must follow the same path. In this case, the messages are relayed through any empty queue

of queue processes never decreases regardless of the fluctuations in the actual queue size, these implementations are not efficient in their use of space. Since an unbounded number of messages are needed to relay messages through the queue, these algorithms are also inefficient in work.

Other researchers implemented unbounded data structures similar to the queue, but they often modified the problem specification to solve a restricted case. Notably, in Agha's Actor implementation of a stack [1], values are pushed and popped based on the *arrival* order of PUSH and POP requests from other processes. However, message order is not preserved in Actor systems, so messages sent between pairs of communicating actors may be received in any order. In Agha's stack, a lone producer-consumer process can issue a PUSH and then a POP, yet not receive the value it had just pushed. Consequently, although the operation of the distributed stack is indistinguishable from the operation of a single-process stack (the nondeterminacy of message arrival order also affects the single-process case), this solution is not a general LIFO structure. This solution is only LIFO from the point of reference of the stack itself, not from the point of reference of any user processes.

Nondeterminacy, the absence of a rigid execution order, is fundamental to the concurrent execution of operations [10, page 4]. However, excessive nondeterminacy can significantly increase the total amount of work to be performed. Agha describes an actor that could be used to remove unwanted nondeterminacy, namely, a *buffer* actor [1] that creates new actors to buffer messages. Restoring the original message order between pairs of actors would require some explicit representation of the original message order that buffer actors could use to identify and forward desired messages, while buffering "out-of-order" messages. For implementations of structures such as stacks and queues, where the ordering of messages is important, restoring the message order could significantly increase the total amount of work. For these problems, total message arrival nondeterminacy is not "useful" nondeterminacy; the additional actor execution orders do not provide enough benefit to outweigh the cost of the increased message handling.

---

element processes, to the queue manager, and then to the requesting consumer process.

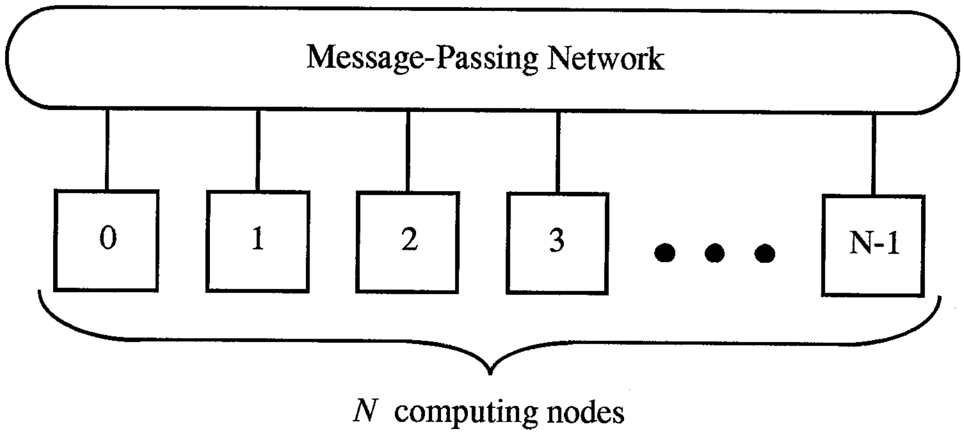


Figure 1.4: Conceptual Model of a Multicomputer. A *multicomputer* is a collection of computers called *nodes* that are connected by a message-passing network. Each of the nodes executes its own program and is composed of a processor and private memory.

1.3 Multicomputers

At first, the unbounded queue problem and its solution may seem to be yet another Computer Science textbook problem and solution under an unfamiliar set of arbitrary constraints. However, the constraints imposed are directly motivated by the architecture of multicomputers.

1.3.1 Architecture

Figure 1.4 represents the conceptual model of a multicomputer [3], a collection of processing *nodes* interconnected by a message-passing network. Each of the nodes is composed of a processor and local, private memory. Each of the nodes executes its program independently (Multiple-Instruction, Multiple-Data). Since the memory on each node is not accessible to other nodes, message passing is the mechanism for internode communication and synchronization.

Figure 1.5 illustrates the niche of the multicomputer architecture in the concurrent

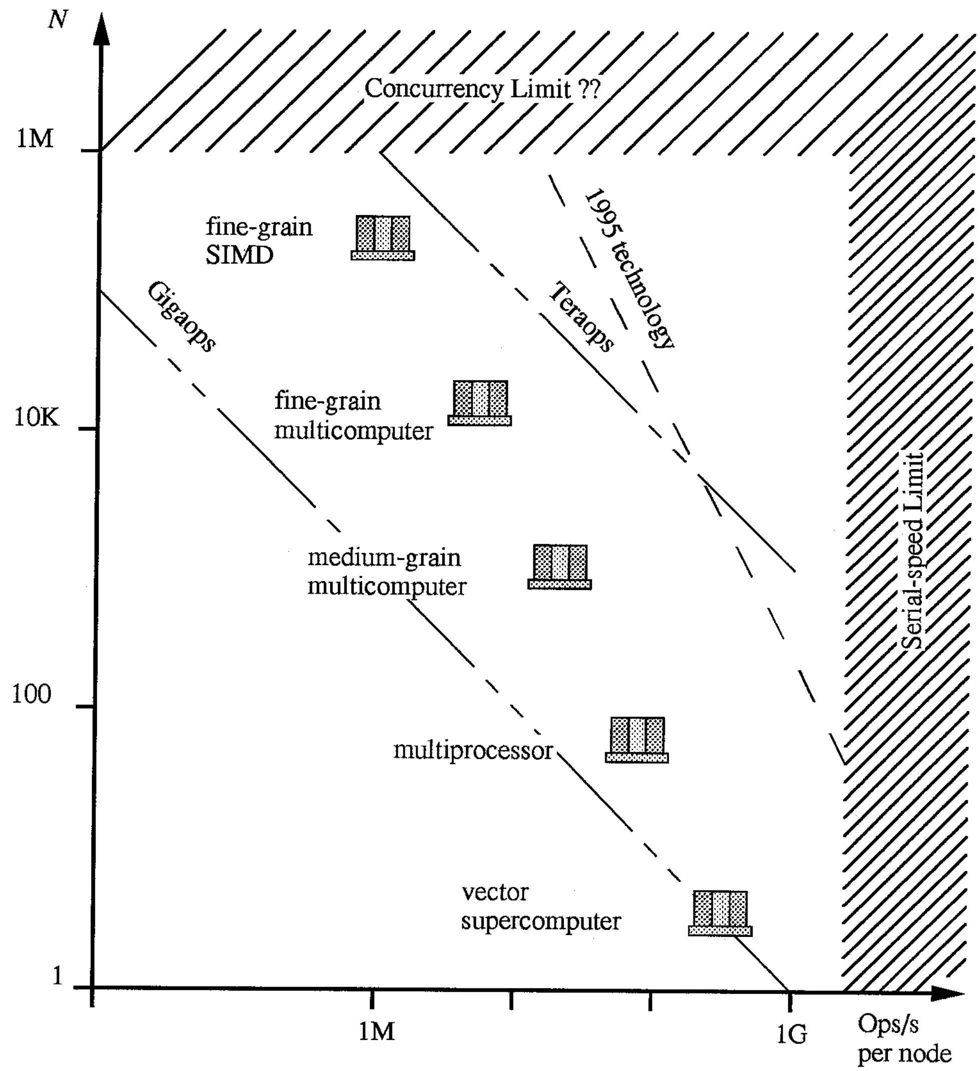


Figure 1.5: Design Space of Parallel Computers. Parallel computers that can be built using one square meter of silicon in 1989 technology (1 GB of primary memory) plotted operations/second versus N, the number of computing nodes. Gigaops and teraops performance can be achieved by increasing the operations/second and/or increasing N. As technology improves, additional hardware resources can be purchased within the same silicon budget. (Figure courtesy of C.L. Seitz)

	Year	System	Number of Nodes	Memory per Node	Processor Speed	Network Channel Speed
medium-grain	1982	Caltech Cosmic Cube	64	128 KB	.15 MIPS	.5 MB/s
	1985	Intel iPSC/1	128	512 KB	.6 MIPS	1.2 MB/s
	1987	Intel iPSC/2	128	4-8 MB	5 MIPS	2.8 MB/s
	1988	Symult S2010	256	4-8 MB	5 MIPS	28 MB/s
	1990	Intel Touchstone Delta	512	16 MB	25 MIPS	30 MB/s
fine-grain	1992	Caltech Mosaic C	16,384	64 KB	11 MIPS	60 MB/s

Table 1.1: Evolution of Medium-Grain Multicomputer Systems. The increases in processor speed and the network channel throughput are consequences of organizational improvements in addition to advances in technology [30, page 157]. The increase in the amount of memory per node has consumed much of the additional technology resources made available as multicomputers evolved. The Mosaic C clearly follows a different scaling track from the medium-grain multicomputer architecture implementations.

computer design space. The architectures plotted represent the approximate configurations that can be built using 1 square meter of silicon in technology available in 1989. (For a full analysis of this argument, see [30, pages 142–147].) As technology advances, a fixed silicon budget purchases more machine resources. The peak performance of a concurrent machine is the product of N, the number of nodes, and the operations per second on each node. Peak performance can be improved if additional resources provided by technology improvements are applied to increasing operations per second and/or the number of nodes. The sequential operation of an individual node is a familiar execution model, so increasing the operations per second has little effect on programming models. However, increasing the number of nodes can require unconventional programming models.

Scaling Tracks

The design choices concerning how to apply additional technology resources have defined two scaling tracks of multicomputer architecture. Table 1.1 details the characteristics of some implementations of the multicomputer architecture that have been constructed during the past decade [28, 25, 2, 31, 22]. The increases in processor speed and the network channel throughput are consequences of organizational improvements in addition to advances in technology [30, page 157]. However, the increase in the amount of memory per node for the so-called *medium-grain* multicomputers has consumed much of the additional technology resources made available as multicomputers evolved. From the Cosmic Cube to the Delta, there has been a *128-fold* increase in the amount of memory per node. The motivation for nodes loaded with memory (often called *fat* nodes) is that these nodes resemble conventional workstations and can be readily programmed using conventional techniques.

Configurations of these multicomputers *under a constant-cost assumption* are plotted in Figure 1.6. The amount of memory per node is a significant factor in how many nodes can be constructed within a fixed budget. The largest Cosmic Cube constructed contains 64 processing nodes, each with 128 KB of memory and capable of 100K operations/second (using 1982 technology). If this machine had been sold commercially, each node would have cost about $4K. Using a fixed budget of $4M, a configuration of 1000 nodes could have been constructed. For the Intel Delta, the complexity of each node (including up to 16 MB of memory) dictates that the same fixed budget could be used to construct a machine with only about 128 nodes (each node is about $30K). Configurations of the other multicomputers in Table 1.1 can also be constructed. (iPSC/1, iPSC/2 and Symult S2010 nodes each cost approximately $10K.) Using the constant-cost assumption, it is clear the scaling track represented by the medium-grain multicomputers in Table 1.1 does not lie in the direction of increasing peak performance. Advances in technology during the evolution of these medium-grain multicomputers have been used to increase the complexity and available memory of the individual nodes rather than increase the number of nodes.

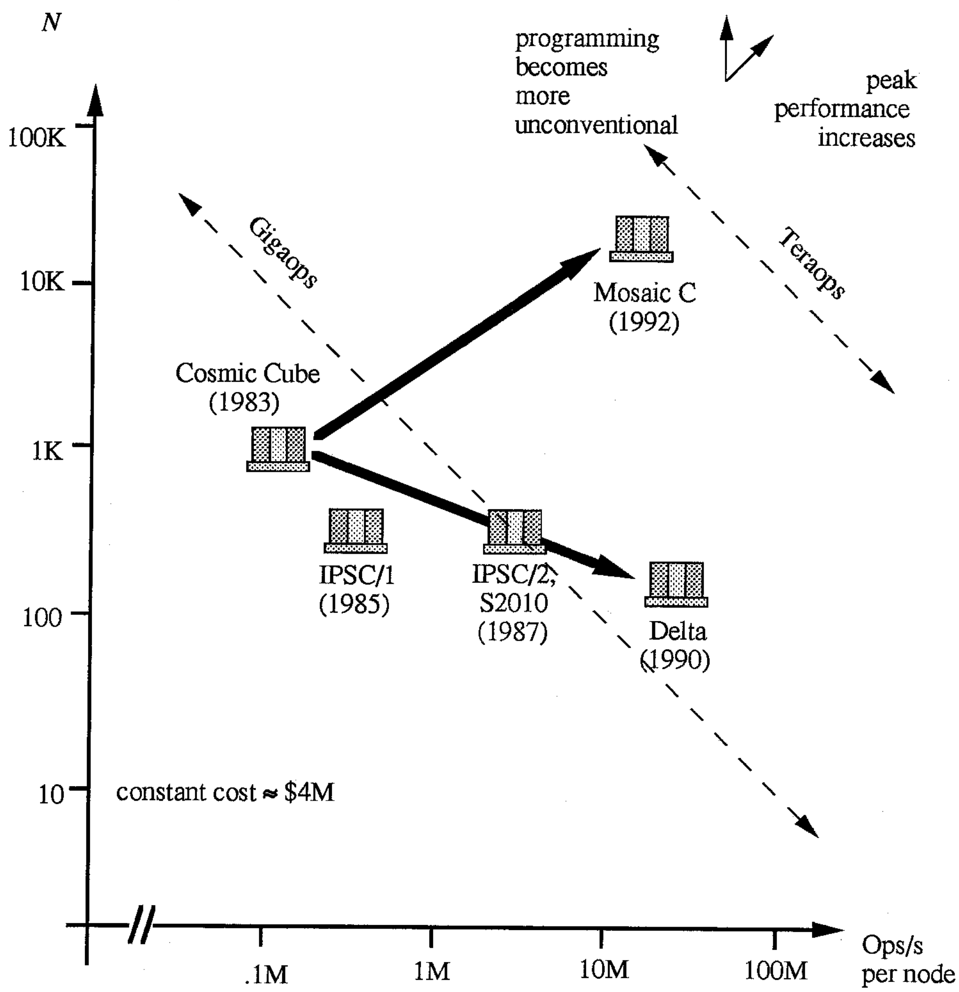


Figure 1.6: Multicomputer Scaling Tracks. The multicomputer architectures illustrated in Figure 1.1 are plotted under a fixed dollar budget (about $4M). The amount of memory per node has a significant impact on the number of nodes that can be purchased within the budget. The direction of increasing peak performance is shown, along with the direction in which application programming becomes more unorthodox.

Fine-Grain Multicomputers

The other scaling track in Figure 1.6 reflects architectures in which new hardware resources are devoted to obtaining additional concurrency rather than additional node performance or memory[30, page 146]. From an initial multicomputer configuration such as the Cosmic Cube in Figure 1.6, if the number of nodes is increased while the amount of memory per node is held constant, the resulting configuration is a *fine-grain* multicomputer. The "granularity" of a multicomputer architecture refers to the amount of resources on each node and the total number of nodes in the ensemble. A medium-grain multicomputer is comprised of a few hundred nodes with several megabytes of memory per node; a fine-grain multicomputer consists of several thousand nodes, each with a few tens of kilobytes of memory. The Caltech Mosaic C multicomputer in Table 1.1 (described in Chapter 2) is an implementation of the fine-grain architecture.

For a medium-grain and a fine-grain machine to have the same peak performance levels, the increased number of nodes in the fine-grain machine must be offset by a proportionate decrease in the operations per second of the fine-grain node. However, the cost of the processor is a relatively small fraction of the cost of a node, so fine-grain nodes can contain processors with roughly the same performance as processors in medium-grain machines. The increased number of nodes in a fine-grain machine provides not only the potential for increased concurrency, but also higher aggregate performance and improved performance/cost.

1.3.2 Programming

Evolution

The evolution of multicomputers[2] has been driven by the goal of improving the performance and performance/cost of parallel computers. The development of hardware, although at times technologically challenging and often innovative, does not represent the critical path

[2]Some parts of the description of the evolution of multicomputer programming notations and their implementations were based on discussions with my advisor, C.L. Seitz, who was involved in the development of most of these programming systems.

Year	Notation		Target Multicomputers	RTS Implementation	
1982	Simula	[21]	fine/medium	*not implemented*	
1985	Cosmic Cube C	[40]	medium	Cosmic Kernel	[40]
1985	CST	[13]	fine/medium	*not implemented*	
1986	Cantor	[4]	fine	Cantor	[4]
1986	Cosmic C	[32]	medium	Reactive Kernel	[33]
1989	Reactive C	[39]	fine/medium	Reactive Kernel	[33]
1992	Affinity	[36]	medium	Affinity Kernel	[36]
1993	C+-	[34]	medium fine	C+- MADRE	[34]

Table 1.2: Evolution of Multicomputer Software Systems.

to achieving this goal. Rather the development of software – programming notations and the runtime support systems executing on nodes– has been the critical effort. As the memory on individual nodes decreases, it becomes less practical to program nodes as conventional computers. As the number of nodes increases into the tens of thousands, conventional programming models and methods are unable to harness the available concurrency.

Throughout the development of new programming notations and runtime systems, the expressive power of the notation and the efficiency of the runtime execution of the program must be carefully balanced. A very expressive, high-level, programming notation usually leads to a commensurately inefficient implementation. Conversely, a very efficient implementation typically relies on limiting the expressive power of the notation [4, pages 155-159].

In this section, the programming models and notations that have been developed specifically for multicomputers will be briefly discussed. In the next section, the runtime systems for executing programs on multicomputers will be described.

Process Model Table 1.2 illustrates the multicomputer programming systems that have been developed within our research group. Each of these system is based on a process model. A process executes its program independently of other processes and maintains a private memory: a process is thus an abstraction of a multicomputer node. By mapping processes to nodes, concurrency can be exploited. If the nodes support multiprogramming (*ie*, multiple processes per node), a degree of machine independence can be achieved.

Simula An extension of the early object-oriented language Simula (1982) [21] was one of the first multicomputer programming notations. By extending Simula to include representations for message passing and concurrent processes, Lang demonstrated that programmers can expose *at the language level* the available concurrency in an application. However, the complexity of the base language, Simula, precluded efficient parallel implementation. Runtime support for this notation was confined to sequential computers.

Cosmic Cube C The language C, extended with primitives for passing messages and creating/destroying processes dynamically (1984) [40], was the first multicomputer programming notation to be used extensively. It was supported on many first-generation multicomputers (Cosmic Cube, iPSC/1, NCUBE) in conjunction with the Cosmic Kernel node operating system (see next section). Since the nodes of early multicomputers were nearly conventional computers, the choice of a conventional programming notation such as C provided a suite of established languages, programming techniques, and tools.

The behavior of each process is defined by a program. These process programs are separately compiled; thus, each process operates within a private address space. Within a program, a process may call primitive functions to create new processes or to send/receive messages. The programmer explicitly controls all process placement by specifying a physical node number and a process identifier during process creation. Messages may contain a user-defined type field; each process can specify which type of message to receive next, based on the process state and the type of the incoming message. If an acceptable message is not available, the process can block (*ie*, not execute additional instructions) until an acceptable message is received.

Reactive-Process Model As experience grew with using this approach to multicomputer programming, it became clear that, for programs engaged in a significant amount of message passing, the arrival of a message was actually the catalyst for the process to execute. These observations were influenced by the work of Agha [1]. Consequently, later programming models focused on computations composed of *reactive* processes.

A reactive process is a process that is normally at rest. It executes for *a bounded time* in response to receiving a message and then either prepares to receive another message or exits. During execution, the process may send messages, create new processes, and modify its internal state. In the earlier process model, a process could specify which message to receive next. A purely reactive process cannot receive incoming messages selectively; it must consume the message at the front of its message queue.

CST The programming notation CST (Concurrent Smalltalk) by Dally (1985) [13] was an early reactive-process programming language. At the time of its development, CST was primarily a reference language. The base language Smalltalk complicated the implementation of parallel runtime support, so CST was supported only on sequential computers. CST now follows its own evolutionary track [12].

Cantor Between 1984 and 1987, Athas developed Cantor, an experimental Actor language [1] that our research group used to study the programming limits of reactive semantics. The Cantor Experiment [4, 5, 8] also explored the balance between programming expressivity and efficiency of implementation. In its early form, Cantor was a deliberately restrictive programming language: programmers could not express pointers, internal iteration, or internal data structures. Cantor is a completely machine-independent notation: machine references, such as node numbers, are not named.

In contrast to the C-based languages, the definitions for all Cantor process types appear *lexically together*. Thus the processes exist in a single address space. However, since there are no pointers, each process can operate safely within a private section of that address space. Since the definitions are compiled together, an analog of interprocedural analysis can be used to perform significant error checking on the messages being passed, and on the processes being created, thus decreasing the required runtime support.

Cosmic C The programming notation Cosmic C (1986) [32], the language C extended with *reactive* primitives for message-passing and process creation, is semantically equivalent to Cantor with respect to reactive processes and messages. However, like the earlier C-based notation, process programs are compiled separately, and machine resources are named directly. Cosmic C is currently used as an efficient, reactive-process programming language for most of the multicomputers in Table 1.1.

Reactive C Another C-based language, Reactive C (1989) [39], was developed by Wen-king Su to experiment with processes executing within a single address space. Since the runtime system does not have to support multiple address spaces and address translation, Reactive C is more efficient than the earlier C-based notations. Reactive C contains little syntactic support for multicomputer programming so it is used primarily for building application-oriented software systems and as a target for higher-level programming notations.

C+- Cantor holds several advantages over its C-based contemporaries: concurrency is expressed at the language level, machine resources are not explicitly named, and compiler analysis provides error checking and runtime information. However, since the complexity of the implementation of the Cantor system does not facilitate experimentation, Seizovic has developed a programming notation derived from C++ called C+- (1992) [34]. Basing a notation on an established language provides a suite of programming tools that can be tailored for experimentation. With its constructs for member functions and classes, C+- provides much of the abstraction necessary to express reactive semantics while also providing significant error checking at compile-time. The C+- notation is briefly described in Chapter 3 and several example application programs are presented and discussed in Chapter 6.

Affinity The Affinity programming notation and runtime system were recently developed by Steele (1992) [36]. Using Affinity, programmers partition the computation into pieces of code and data. Light-weight, reactive processes called *actions* are executed to provide atomicity and maintain the consistency of data modifications. Shared data structures can be accessed in a shared-memory fashion, but without the need for explicit locking to control access and provide mutual exclusion.

Fine-Grain Programming

Each of the programming notations described above can be used to program any multicomputer. The programming techniques used, however, tend to reflect the underlying architecture. For example, programmers of medium-grain multicomputers may express their applications as collections of tens or hundreds of processes, each with kilobytes of data. These processes infrequently send each other messages that are kilobytes in length. New processes are seldom created; the large cost of creating a new process is weighed carefully against the amount of computation that can be performed by that new process. Applications written to be executed on medium-grain machines may consist of processes that are simply too large even to be loaded on a fine-grain machine.

In contrast, fine-grain computations are typically collections of hundreds, thousands, or tens of thousands of small cooperating processes [8]. Messages are frequently sent between processes; these messages may be only a few words in length. New processes are created liberally; virtually any opportunity for concurrency is exploited. Applications written to be executed on fine-grain machines would be grossly inefficient on a medium-grain machine where the number of nodes is small and context switching times are large; message latencies are much higher; and the cost of creating new processes is prohibitive.

The *light-weight* process model, in which processes with a small amount of state execute within a single address space, is a good fit to the fine-grain architecture. (This model was described as "The Rules of the Game" in Section 1.1. Chapter 3 describes the fine-grain programming methods that are used to express computations using this programming model.) This refinement in the programming model reflects the axiom that as nodes of a multicomputer become more unconventional, the accompanying programming models also must become more unconventional.

1.3.3 Operating and Runtime Systems

EVOLUTION

Table 1.2 lists the operating and runtime systems that have been developed to support multicomputer programming notations. Runtime support on parallel machines for the early programming systems such as Cosmic Cube C [40] was essentially conventional. The Cosmic Kernel [28] was widely distributed as the operating system running on individual nodes of the first-generation multicomputers (Cosmic Cube, iPSC/1, NCUBE). In this system, a process is represented by a code segment and a data segment. The runtime system performs address translation to support a private address space for each process. Processes are scheduled to run by the runtime system using round-robin scheduling. If a process blocks on receiving a message, a full context switch occurs to allow another process to be scheduled to run. As the entire state of the process must be saved, context switches are quite expensive. The runtime system also uses timers to ensure fairness in process execution, swapping out long-running processes. Intel's NX node operating system (1987) [26] was (and is) based on this runtime model.

The next generation of runtime systems adopted the reactive property as its primary scheduling mechanism and provided support for reactive processes. The Reactive Kernel (1988) [33] is a widely distributed node runtime system that is based on reactive processes. (The Cosmic Environment [39] is the companion host runtime system.) Processes are represented by a code segment and a data segment, but process scheduling is dictated by the queue of messages entering the node. The runtime system *executes* the message queue, invoking the process that is the destination of the message at the front of the message queue and then removing that message from the queue.

The Cantor runtime system [4, 5] is also based on reactive scheduling. Cantor aggressively exploits compiler technology to minimize runtime support. In addition, this system automatically manages system resources, including placing processes. A series of simulations [4], conducted using a simplified machine model, demonstrate that a runtime system can do a reasonable job of automatic resource management.

The Affinity Kernel (1992) [36] is a node runtime system that supports the execution of reactive processes that share data structures.

The MADRE (**M**os**A**ic **D**istributed **R**untim**E**) system has been developed as the runtime system for the Mosaic C. This runtime system incorporates the goal automatic resource management and the reactive-scheduling mechanisms from earlier multicomputer runtime systems. MADRE is described in detail in Chapter 5.

Fine-grain Runtime Systems

Aggressive fine-grain runtime system design is facilitated by the reactive-process computational model. Consider programs where processes do not reactively consume messages. As the amount of memory is quite limited on a fine-grain node, few unwanted messages can be buffered before exhausting the available node resources. When a node can no longer receive messages, additional messages remain queued in the message network, blocking communication channels and potentially causing deadlock in the network. Reactive programs consume messages in the order received hence there are no messages needing to be buffered *indefinitely*. This "consumption assumption" [24] ensures that, even though queued messages may temporarily block communication channels in the network, all messages will eventually be received and consumed by the node, thus avoiding communication deadlock.

In addition, since a reactive process executes (by definition) for a bounded time, no timing mechanisms are needed to ensure fairness in process execution. Runtime support for fine-grain computations can be simplified so that process scheduling is analogous to invoking a function. [30, page 160].

However, the physical realities of a fine-grain node do have significant repercussions for process placement and other runtime resource-management issues. In general, a multicomputer node maintains an internal queue for incoming messages so that messages can be removed from the network as quickly as possible. If, on a medium-grain machine, the memory of a single node is exhausted, due to the programmer placing too many processes on the node or due to overflow of the resources allotted for this incoming message queue, the computation fails. Since there are typically several megabytes of memory resident on the node, such a failure is not likely to occur and can usually be completely avoided by better process placement or communication strategies. However, if the same runtime model is used on a fine-grain machine, a local memory fluctuation of a few *kilobytes* of memory can cause the entire computation to fail. At the point of failure, only a tiny fraction of the total memory of the machine may actually have been exhausted. In addition, since process placement is managed by the runtime system, the failure of the computation may be directly attributable to the automatic placement algorithm. In fact, if the placement algorithm exploits nondeterminism, the failure may not be repeatable on successive executions of the program. Robustness clearly must be a high priority for fine-grain runtime systems.

The small amount of memory on a fine-grain node also precludes the abstraction of infinite memory that is conveniently available on medium-grain nodes. For example, in a medium-grain machine, the code for each process type can be resident on every node without a significant impact on the computations that can be performed. Even if all the code could fit onto a fine-grain node (which it often cannot), devoting such a large fraction of the total memory of the machine to maintaining thousands of identical copies of the code is clearly unreasonable. The fine-grain runtime system must find more sophisticated ways to handle the code replication problem .

These two problems highlight the apparent conflict between the convenient abstraction of a global address space containing infinite memory and the reality of distributed memory. The solutions to these problems stem from the observation that the memory of a fine-grain machine is actually *less distributed* than the memory of a medium-grain machine. As the overhead to send and receive messages decreases, and the performance of the message network increases, the barriers to cooperation between nodes are reduced. Fine-grain runtime systems are distributed operating systems that cooperate heavily to blur the physical boundaries of the nodes.

The principal goal of this thesis is to develop and present a family of algorithms that form the basis for fine-grain multicomputer runtime systems. Included in these runtime systems will be robust mechanisms for automatic resource allocation and memory load balancing, solutions to code replication and large data structure problems. These algorithms, although designed primarily for fine-grain machines, will be applicable to both coarser grain and future multicomputer architectures. The queue result presented earlier is particularly important to this effort since it provides the selective-receive capability not only to the user program, but also to the runtime system, thus permitting more latitude in algorithm design.

1.3.4 Significance of the Queue-Problem Solution

The solution to the queue problem, which enables the queue manager to perform a selective receive (waiting on the reply to a **get** message), illustrates how a fine-grain node can selectively receive messages *without overflowing its small local memory.* In this analogy, the queue manager is replaced by a fine-grain node runtime system, and the data queue by a process. In normal, reactive operation, the node runtime system consumes messages in the node receive queue by delivering the oldest queued message to its destination process for processing. If a process decides to selectively receive a message, that process is, in effect, disconnecting itself from the message delivery operation of the runtime system. This disconnection is analogous to the data queue being disconnected from the queue manager during a **get** operation. Until the desired message arrives and the process is logically reconnected to the runtime system, the runtime system queues unwanted incoming messages by creating remote processes that store the messages as private data. Transforming queue messages into processes uses the only available mechanism for growth of the computation – process creation – to satisfy the message consumption property. Queued messages are later retrieved and delivered in order to the destination process. Thus, we have implemented a selective-receive mechanism using only the reactive semantics that are ideally suited for fine-grain machines.

1.4 Thesis Overview

Figure 1.7 illustrates the various levels of computation for a fine-grain computation. At the top level, a programming agent (either a programmer or a high-level program translation tool) expresses a computation as a set of processes that communicate via messages. At the next level, compile-time analysis and error-checking reduce the support required from lower levels of computation. The runtime system level, the focus of this thesis, is the vital interface between a program and the hardware on which it is to be executed. At the bottom level, machine hardware operates at the level of sending and receiving messages, and executing instructions. In Figure 1.7, the thesis chapter that addresses each level of computation is also indicated. Throughout the thesis, an attempt will be made to separate the fundamental ideas of the programming system, runtime system, and the hardware architecture from their implementations.

The experimental apparatus used to investigate this thesis consisted of a prototype runtime system, an experimental fine-grain multicomputer called Mosaic C, and an experimental programming notation and compiler called C+−. Each of these components was developed by members of C.L. Seitz's research group at Caltech. The chapters on the hardware and programming describe the essentials required for this thesis to be self-contained; full discussion of these topics can be found in the cited literature.

In Chapter 2, we describe the essential characteristics of the Mosaic C. Each node of this multicomputer is implemented as a single VLSI chip, and the composite machine contains 16K nodes. C.L. Seitz, Jakov Seizovic, Don Speck and Wen-king Su are the primary members of the design team. The software-development tools that were used to develop and profile the prototype runtime system are described.

Chapter 3 presents a synopsis of the programming notation C+−, designed and implemented by Jakov Seizovic. This notation is used throughout this thesis to express appli-

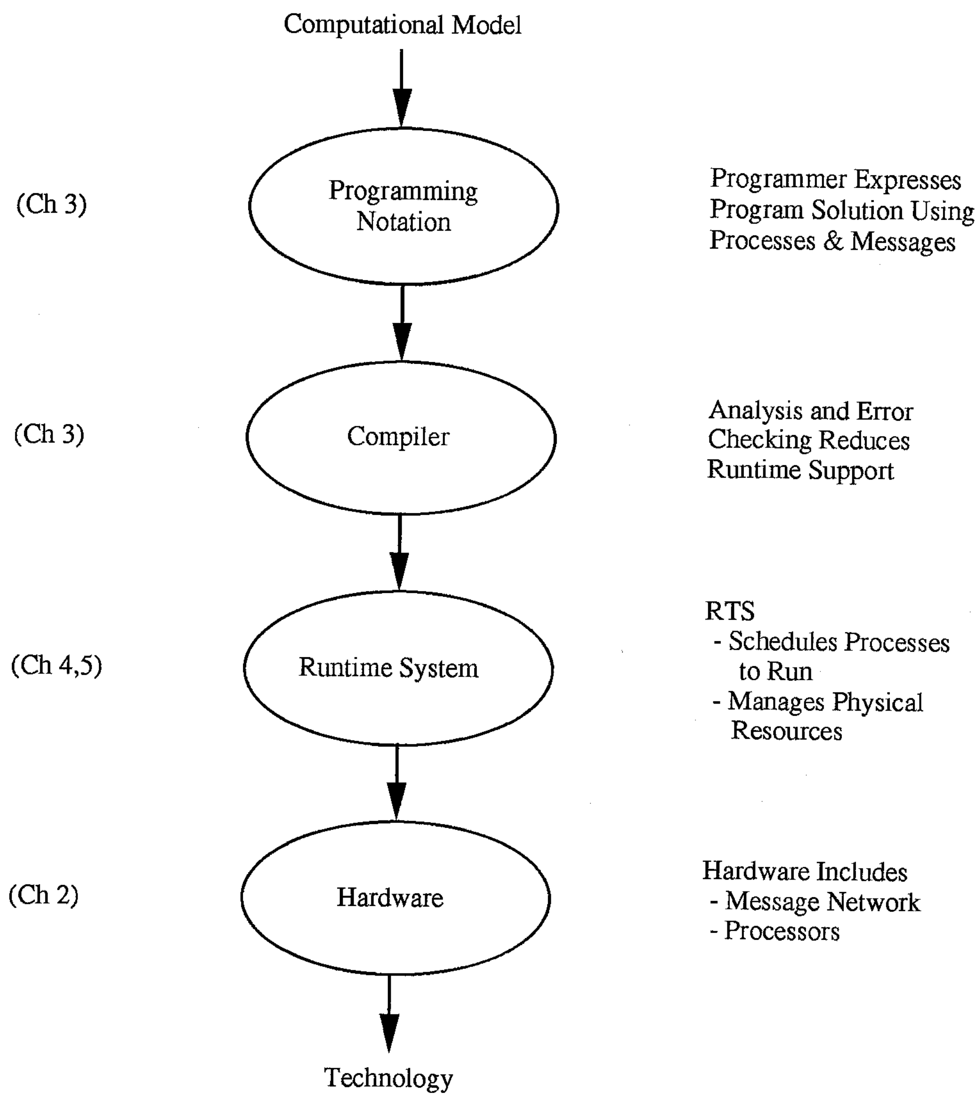


Figure 1.7: Levels of Computation. The runtime system level, the focus of this thesis, is the vital interface between a program and the hardware on which it is to be executed.

cation programs. Several fine-grain programming methods that have evolved for fine-grain multicomputers are also presented.

In Chapter 4, the runtime system design criteria used in this thesis are outlined. We present a method for expressing fine-grain runtime systems as fine-grain programs.

Chapter 5 describes a complete runtime system for Mosaic ensembles, which I designed and implemented using C+−. This prototype runtime system is itself composed of fine-grain processes that are distributed across the nodes of an ensemble. Components of the runtime system cooperate to provide a robust environment for the efficient execution of application programs. Several alternative runtime algorithms for resource management are discussed in detail. The modular design of the Mosaic runtime system permits substitution of these alternative runtime algorithms.

Chapter 6 presents several application programs that I developed to assess the algorithms and performance of the runtime system. The application programs were chosen to represent expected programming patterns so that we could identify which runtime algorithms perform best for different classes of programs. Experiments were run with different runtime system algorithms on various configurations of Mosaic nodes. An evaluation of the resource management algorithms in Chapter 5 is presented.

Chapter 7 ends the thesis with an assessment of the contributions and future directions of this thesis work.

The execution of fine-grain programs and the efficient utilization of the physical resources of fine-grain ensembles present special challenges to a runtime system. The contribution of this thesis is to demonstrate that a runtime system can meet these challenges. The overall success of the fine-grain multicomputer architecture depends upon the successful implementation and integration of application programs, the runtime system, and hardware.

2 The Mosaic Project

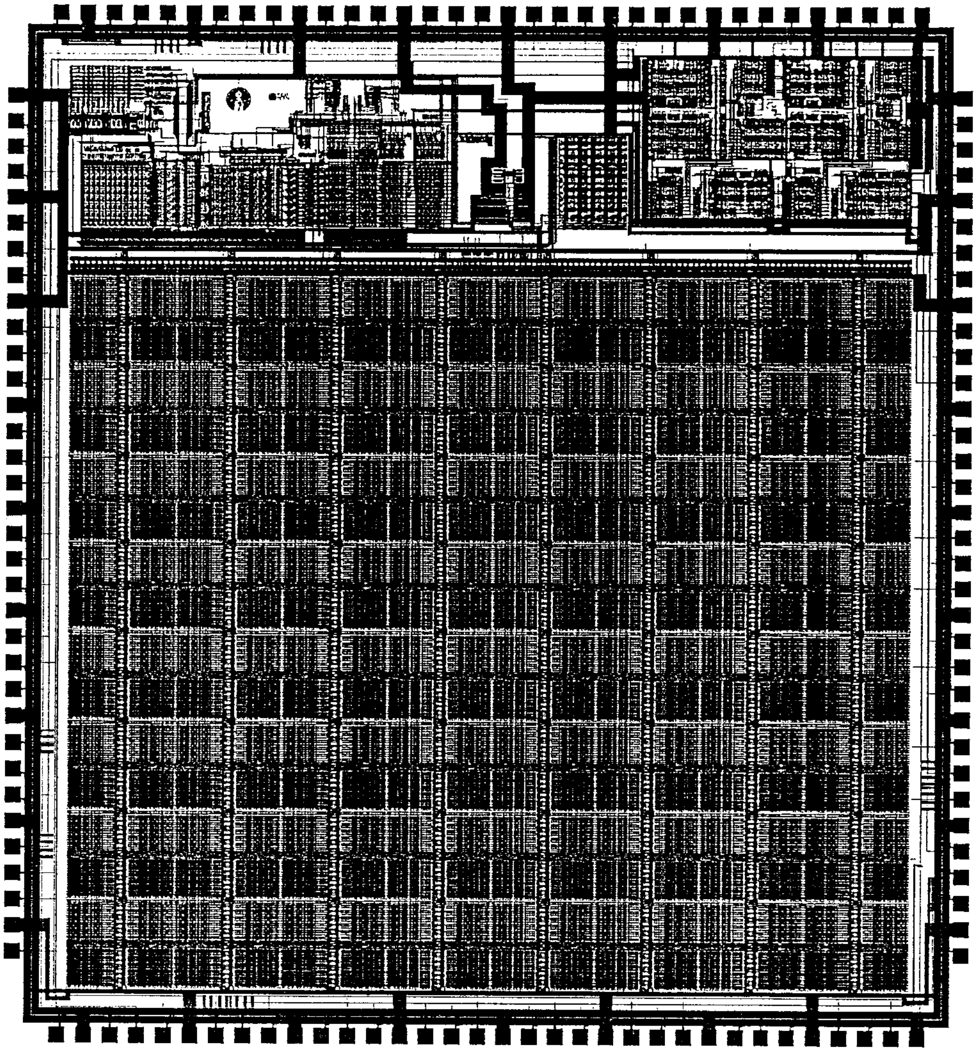
By definition, the fine-grain multicomputer architecture consists of many small nodes and permits a range of implementations. Our research group at Caltech is implementing a fine-grain multicomputer called Mosaic C. The goal of this project is not only to develop a high-performance machine, but also to develop the experimental apparatus for investigations into other issues for fine-grain machines. For example, the Mosaic is a target for the Program Composition Notation (PCN) of Chandy and Taylor [11], a high-level architecture-independent concurrent language. The Mosaic is also the experimental platform for the series of runtime experiments presented in this thesis. (For a description of the design and implementation of the Mosaic, see [9]).

As depicted in Figure 1.4, the multicomputer architecture can be decomposed into its constituent parts, the computing node and the message-passing network. It is the implementation of these two parts that defines the overall multicomputer implementation.

2.1 A Mosaic Node

A Mosaic node is a self-contained computer implemented as a single VLSI chip. Each node includes a processor, random-access memory(RAM), and read-only memory (ROM). In addition to these conventional elements, the node contains a router and a packet interface. A shared bus connects the processor and the packet interface to the RAM and ROM, and *vice versa.* The router communicates with the node only via the packet interface. Similarly, the other nodes in the network communicate with the node only via the router. Each of these components is discussed further in following subsections.

The VLSI layout of the Mosaic node is shown in Figure 2.1; Figure 2.2 identifies the major components in the layout. This layout was developed entirely within our research



Actual size

Figure 2.1: Layout of a Mosaic Node. The actual size of the chip when fabricated is 9.2mm by 10mm.

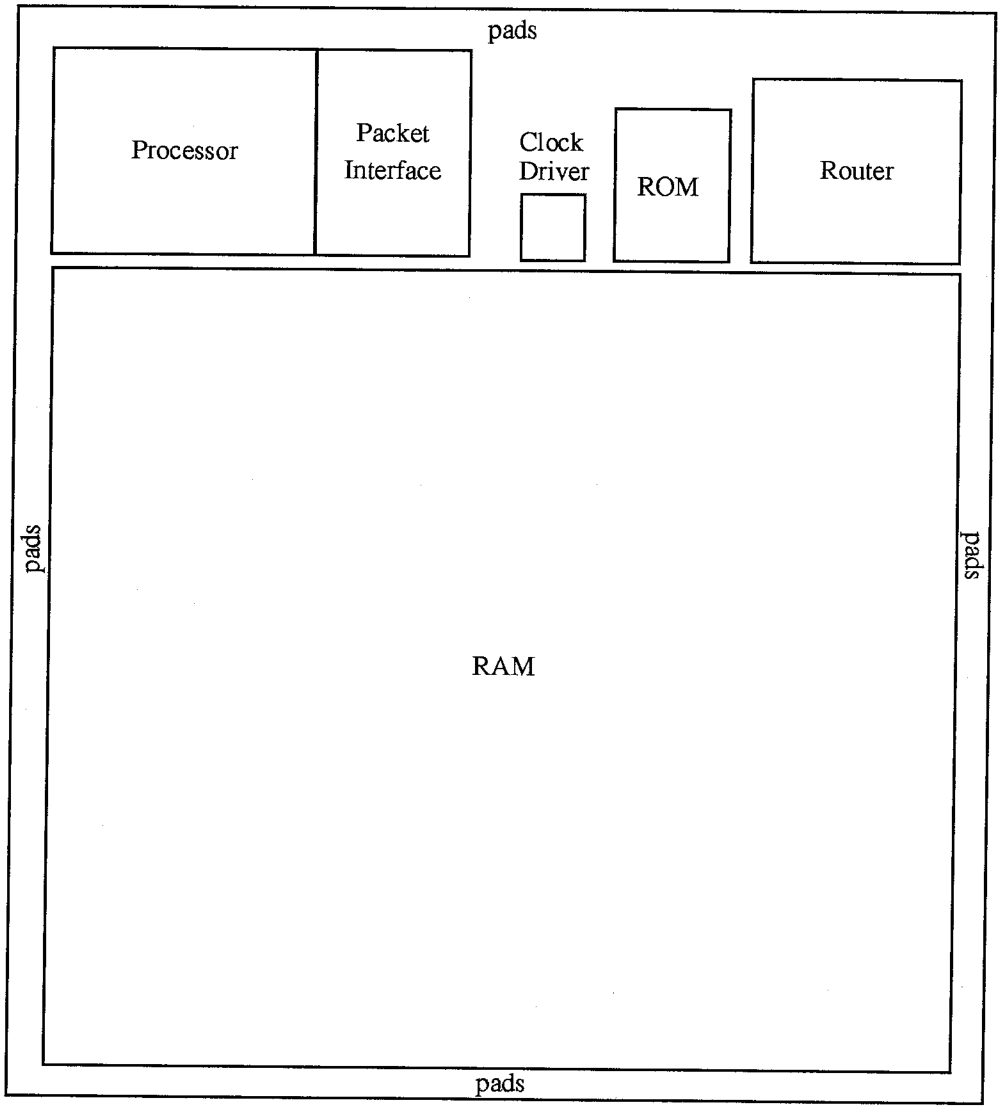


Figure 2.2: Mosaic Node Components. Each node includes a processor, random-access memory(RAM), read-only memory (ROM), a router and a packet interface.

group [34].

Implementing nodes as single-chip computers has been a goal throughout the multicomputer evolution. In [28] (1985), Seitz says

> "The Cosmic Cube nodes were designed as a hardware simulation of what we expect to be able to integrate onto one or two chips in about five years".

Table 1.1 illustrates the similarity between the configurations of a Cosmic Cube node and a Mosaic node.

2.1.1 Processor

The processor design of a Mosaic node is essentially conventional. The 16-bit, integer, microcoded processor is rated at 11 MIPS using a 30 MHz clock. One of the unconventional aspects of the Mosaic processor is that it is a two-context processor. In other words, the Mosaic processor includes two independent program counters. Nominally, one program counter is used for executing system programs, the other for the user's computation. The two contexts, SYSTEM and USER, provide the illusion of two processors (one with priority) working in the same address space.

The register file of the processor reflects the dual-context nature of the node. The processor has 24 data registers – 8 private to each context, and 8 that can by accessed by both contexts. Including the two program counters, there are 16 address registers that are accessible by both the SYSTEM and the USER contexts.

2.1.2 Random-access Memory

A computation is insensitive to message latency that is less than the interval during which the message would otherwise have resided in the node receive queue. Thus, the latency of interprocess communication (*ie*, message passing) can be at least partially hidden using multiprogramming [30, page 195]. In addition, since message passing typically transfers blocks of data at once, the cost per data item is small. However, the latency of intra-process communication (*ie*, processor-to-memory communication) cannot be hidden or amortized.

The processor becomes idle if memory access time is more than about one instruction time. With the processor and memory on the same chip, the latency of processor-to-memory communication for Mosaic is reduced.

In addition to being fast, the Mosaic memory must also be dense, given that all the memory is resident on the chip. The Mosaic RAM cells were designed to achieve both goals [35]. The RAM is composed of 1-transistor cells, with a total of 64 KB of memory. The access time is 2B per cycle (every 33 ns running at 30 MHz). Since the memory is fast relative to the speed of the processor, large register files or sophisticated register allocation are not warranted. Accesses to memory are nearly as fast as register accesses.

2.1.3 Read-only Memory

In addition to RAM, the Mosaic node includes 1 KB of ROM. This memory contains self-test and bootstrap code. Self-testing is particularly important for fine-grain computers since there are thousands of nodes, and most of the complexity is internal to a node.

Upon reset and successful self-test, the node begins executing the bootstrap code in the SYSTEM context. This code configures the node so that a program can be loaded. The bootstrap code

- configures interrupts to receive a message,
- configures message-pointer registers so that the incoming message will be written beginning at the lowest RAM address,
- when the message arrives, the program counter jumps to the lowest RAM address and begins executing instructions.

(See next section for description of Mosaic interrupts and message pointer registers.) The initial message contains the runtime system, or whatever code is to be executed in the SYSTEM context. By setting the APC (Alternate Program Counter) register, the code to be executed in the USER context is specified.

The Mosaic ROM does not presently contain the runtime system. This decision is based on the desire to experiment with different runtime system algorithms. In future implementations, established runtime algorithms could be moved to the ROM.

2.1.4 Router

The unconventional components of the runtime system are those concerned with message passing, the router and the packet interface. The router design in the Mosaic is an oblivious, dimension-order router as described in [16]. The router on each node has eight channel connections to its neighbors (bidirectional in each compass direction). Each channel consists of 8 data lines, a line to indicate that the current byte is the last byte in the message (*ie*, tail bit line), and lines for data request and acknowledge (used for flow control). The channel connections of the router require 88 pins and are in fact the only external logical connections of a Mosaic node.

2.1.5 Packet Interface

The packet-interface logic within the node controls the flow of messages between the node and the message network. The packet-interface unit and the processor communicate to send and receive messages via interrupts. The packet interface transfers incoming messages from the router directly into the memory using Direct Memory Access (DMA) logic. Two address registers denote the boundaries of the receive buffer – the Message Receive Pointer register (MRP) and the Message Receive Limit register (MRL). The packet interface writes a word at the MRP location and then increments MRP. If the entire message is copied into the provided buffer, *ie,* the tail bit of the message is observed by the packet interface, a **receive_interrupt** is generated. If the buffer is exhausted (MRP == MRL), a **buffer_full_interrupt** is generated. When the **buffer_full_interrupt** is acknowledged indicating that a new buffer has been allocated, the packet interface resumes transferring the incoming message using the new values of MRP and MRL.

Similarly, the packet interface uses DMA access to transfer outgoing messages from the node memory, through the router, into the network. The buffer to be injected into the network is demarcated using two address registers – the Message Send Pointer register (MSP) and the Message Send Limit register (MSL). The processor indicates that the message is ready to be sent by writing the relative coordinates of the destination node in the Delta-X, Delta-Y address register (DXDY). The packet interface transfers the word at the MSP location to the router and then increments MSP. When MSP equals MSL, the entire message has been sent so the tail bit is sent and a **send_interrupt** is generated.

Two additional address registers are used to manage interrupts. IMR, the Interrupt Mask Register, is written by the processor to enable or disable each of the three interrupts. If an interrupt is generated by the packet interface, but the IMR indicates that that interrupt is disabled, the interrupt is queued. ISR, the Interrupt Status Register, represents the set of pending interrupts, one bit per interrupt. To acknowledge interrupts, the processor writes back the values of the interrupt bits.

The effect of an interrupt differs depending on the context of the processor. If the processor is in the USER context and the interrupt is enabled, a context switch occurs. Now executing in the SYSTEM context, the processor is expected to handle, and then acknowledge, the interrupt. A machine instruction called **punt** is executed to resume processing in the USER context. If the processor is executing in the SYSTEM context, enabled interrupts can be set but no context switch occurs. When the processor switches to the USER context, these pending interrupts may trigger an immediate context switch to the SYSTEM context.

Nominally, Mosaic interrupts are used in the following manner. To send messages, the processor queues outgoing messages with their destinations. If no **send_interrupt** is pending, the processor sets MSP and MSL and then triggers the sending of the message by writing DXDY. (Note that writing these three registers constitutes the minimum software overhead for sending a message.) When the processor is notified via the **send_interrupt** that the message has been sent, the buffer containing the sent message can be freed and the sending of the next outgoing message is initiated. To receive messages, the processor must first indicate, using MRP and MRL,[1] the area of memory where an incoming message

[1]MRL actually points to the word *after* the last word that should be written. Thus, when the last word

should be written. Upon receiving a **receive_interrupt**, the processor can queue the message or deliver it immediately to the destination process. If a **buffer_full_interrupt** is received, the processor is expected to allocate a new receive buffer, reconstruct the message that may have been fragmented, reset MRP and MRL, and acknowledge the interrupt so the remainder of the message can be received. However, the simple design of the Mosaic interrupt system provides great flexibility in runtime approaches to message handling. In section 5.2.2, we discuss an aggressive use of the Mosaic interrupt mechanism for receiving messages.

2.2 The Mosaic Message-Passing Network

The Mosaic interconnection network is logically a two-dimensional, bidirectional mesh. Buffering within the channels provides asynchronous message passing between nodes. Messages travel along communication channels within the mesh using algorithmic, cut-through routing. Messages travel first along the x-dimension of the mesh, and then along the y-dimension until the destination node is reached.

When a message is sent, the packet interface on the sending node prepends a message header containing the direction (positive or negative) and distances in each dimension to the destination node. The router of each node along the message path examines the message header; non-zero distances in the dimension being routed are decremented and the message is routed to the next node in that dimension. If the x-component is zero, the message is routed into the y-dimension. If the y-component is zero, the message has reached its destination. This routing algorithm preserves the FIFO order of messages sent from a source node to a particular destination node.

The choice of a light-weight process model (section 1.3.2) impacts the throughput and latency requirements for message network implementations.

Medium-grain computations are expressed as collections of processes that communicate infrequently, using large messages. Since a fine-grain computation is partitioned into many small pieces, those pieces are more interdependent, and thus communicate more fre-

---

has been filled and MRP incremented, the **buffer_full_interrupt** is generated.

quently, using smaller messages. In addition, the message load on a fine-grain message network can be heavier, simply because more nodes can send messages concurrently.

A fine-grain computation is more sensitive to message latency since, due to the increased concurrency of message consumption, messages stay enqueued at a node for less time [30, page 195]. Using efficient network hardware and low software overhead for message passing (section 2.1.5), the message system of the Mosaic exhibits latencies more than one order of magnitude less than medium-grain systems.

The emphasis on message-passing performance can be observed by comparing the network capabilities of the Mosaic and the Intel Delta [22]. The communication channels for both machines are essentially identical except for their throughput rates: Mosaic channels run at 60 MB/s, Delta channels at 80 MB/s. The Mosaic possesses a significant architectural advantage over the Delta however: the speed of the Mosaic processor and memory (30 MHz) is roughly 60 MB/s. This speed matching of the node and the communication network means that *the node can source and sink at the network rate.* In contrast, a Delta node can receive or inject messages into the network at only 30 MB/s. For message sending, the Mosaic can inject a word of the message per cycle. The Delta however can inject a word only on every third cycle. By not injecting packets during the two other cycles, the effective length of the message is trebled. Similarly, for receiving messages, the Mosaic can remove messages from the network faster, improving throughput and reducing latency. In comparison with medium-grain machines, the speed matching of the Mosaic node to the network rate provides a significant improvement in communication performance.

2.3 Mosaic Ensembles

Mosaic nodes are currently being fabricated using a 1.2μm, 2-level-metal, Hewlett-Packard CMOS process. Each chip contains about 1.2M transistors, mostly located in the RAM. The typical yield from prototype runs has been roughly 40%, resulting in an unpackaged cost per node of about $30.

A board of 64 Mosaic chips, arrayed 8$\times$8, is the fundamental unit of construction of

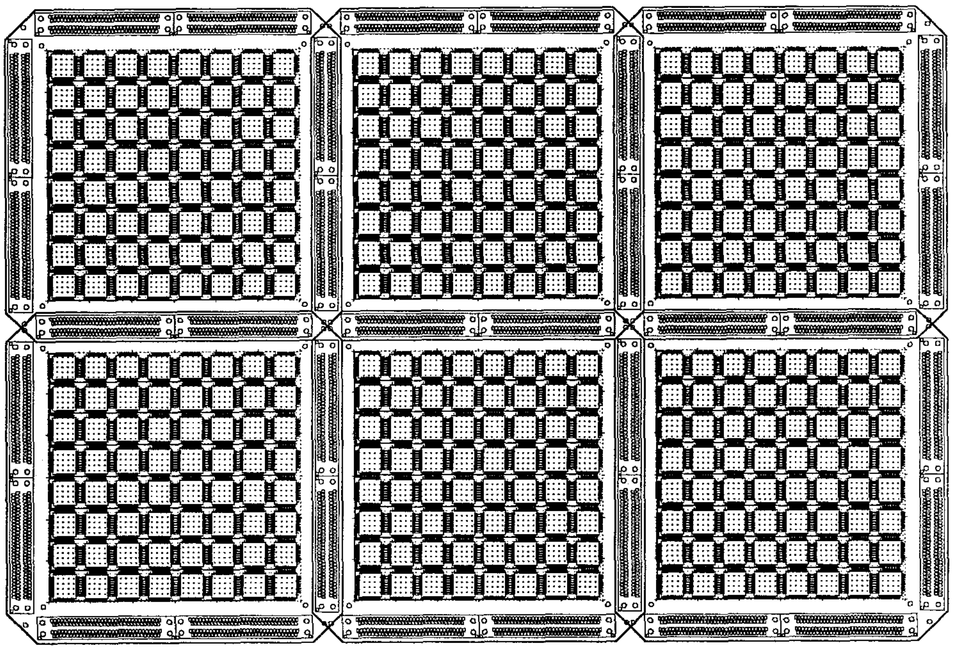


Figure 2.3: 8×8 Mosaic Array Boards. Mosaic boards self-compose using stack connectors so that arbitrary two-dimensional configurations can be assembled.

Mosaic ensembles (Figure 2.3). Mosaic chips are packaged using tape-automated bonding (TAB) technology. TAB packaging differs from conventional packaging in that chips are bonded directly onto boards. This approach provides improved chip density: the size of the 8×8 array is only 6 inches square; and reduces the cost of packaging: the cost of each packaged node is about $75, or $4800 per board.

Mosaic boards self-compose using stack connectors so that arbitrary two-dimensional configurations can be assembled. The largest ensemble of Mosaics used for the experiments reported in this thesis is a 16×16 array (4 8×8 boards). Chips are currently being fabricated to construct an ensemble of 128×128 nodes (16,384 nodes).

2.4 Host-Interface Boards

The first Mosaic-C node prototypes were fabricated as "memoryless" nodes in 1990. These nodes included the processor, packet interface and router, but no RAM or ROM. The motivation for this prototyping decision was twofold. First, debugging the Mosaic processor was simplified because the memory addresses it emits could be examined directly.

Second, these nodes can be mounted, along with commercially available dRAM, onto circuit boards to construct a *Mosaic Development Board.* By plugging these boards into a conventional workstation, the external memory is readable and write-able by both the workstation and the Mosaic processor. Because this shared memory simplifies debugging and monitoring of code (see section 2.5.2), several small ensembles (< 32 nodes) of Development Boards were constructed to facilitate program development.

A Development Board is used as the host interface between the workstation and the Mosaic ensemble. The host-interface configuration currently used for Mosaic ensembles includes 2 memoryless-Mosaic nodes mounted on a Sun SPARCstation Sbus board. Each node includes 64K×16 external memory. The Sbus board is connected to the Mosaic ensemble using cables and custom "slack" chips that execute a zero-slack (non-interference) protocol [27].

2.5 Programming Toolkit

2.5.1 Compilers

The low-level programming toolkit for the Mosaic C includes a C compiler and a C++ compiler. Each of these compilers was developed by porting the corresponding Gnu compiler to the Mosaic. A complete Pascal programming system, including compiler and runtime system, has been developed by van de Snepscheut and his students [23].

2.5.2 Debugger

An assembly-level debugger is available for use with Mosaic Development Boards. The interactive user interface for this tool allows the following debugging operations to be executed on some or all of the nodes in the ensemble:

- **stop** *address*

  Since the program being debugged is loaded into memory that is readable/write-able by the workstation, the instructions at the location *address* can be overwritten with instructions that cause the processor to jump to a special **dump** routine. This routine causes the values of the registers to be written to a reserved region of memory. The execution then enters a loop, awaiting a command to resume program execution, *eg*, **run** or **step**.

- **run**

  When this command is received from the user interface, the original program code that was overwritten is restored, and execution resumes after the most recent breakpoint.

- **step** [*count*]

  count assembly instructions are executed ,or instructions are executed until a breakpoint is encountered, whichever occurs first. The default *count* is 1.

- **print** [<*address*> | <*register*>]

  The values of memory locations and the values of the registers (sampled at the most recent breakpoint) can be examined.

Other debugging commands include **trace**, to control whether the instructions being executed are echoed, **set**, to set the value of a memory location, and **cont**, which causes the program to run until the next breakpoint or the next context switch.

Programs can be debugged in this environment and then moved over to the larger ensembles constructed from 8×8 boards. The Mosaic runtime system discussed in this thesis was developed using this tool.

2.5.3 Host-Interface Routines

The host program running on the workstation communicates with the Mosaic ensemble via the host-interface board (section 2.4). Mosaic ensembles currently use one node on the host-interface board for messages sent from the host to the ensemble (host-sending node), and one node for messages sent from the ensemble to the host (host-receiving node).

The code executed by the host-sending node configures the memory on that node as a circular queue. The host program writes messages into this shared memory, and then writes message-pointer values into pre-defined memory locations to inform the sending node that a message is ready to be sent. The sending node sends the message to the ensemble by writing the MSP, MSL and DXDY registers (section 2.1.5).

The host-receiving node also configures its node memory as a circular queue. The registers MRP and MRL are set so that messages sent from the ensemble are written into the queue by the node's packet interface. When a message has been completely received, the receiving node modifies the message pointers in pre-defined memory locations.

The host program running on the workstation interacts with the sending and receiving interface nodes using a set of message-passing primitives patterned after the *x primitives* used in earlier programming systems such as Cosmic C[32]. The Mosaic primitives are called *m primitives.*

- void *mmalloc(int *size*)

  This function returns a message buffer of size *size*. The length of the buffer is implicitly stored within the buffer.

- void msend(int **message-pointer*)

  The contents of the message buffer pointed to by ***message-pointer*** are copied into the circular queue in the host-sending node.

- void *mrecv()

  This function returns a pointer to the next message sent from the Mosaic ensemble to the host. If no message is pending, this function returns NIL. A variant function, mrecvb(), blocks until a message is received.

3 Fine-Grain Programming Using C+-

As illustrated in the opening of Chapter 1, the "rules of the game" of fine-grain concurrent programming are easy to state. These rules, however, have evolved through the past decade of our research group's investigations of multicomputer programming (section 1.1). Our approach has been first to design a programming notation based on a computational model that appears to have properties favorable to expression and implementation. Then we write enough non-trivial programs in the new notation to evaluate both the expressivity of the notation and the programming techniques that necessarily emerge.

The programming language C+- is the latest in a series of programming experiments using this approach (section 1.3.2). In this chapter, we first present the C+- notation, with particular emphasis on the aspects of the notation used to express programs in this thesis. We then present a general description of the fine-grain programming methods that have evolved in conjunction with the multicomputer architecture.

3.1 C+-

C+- is a C++-based programming notation developed by Jakov Seizovic that will be described in detail in his upcoming Ph.D. thesis [34]. C+- is a translation-based system — programs are written in C+- and then translated into C++. The translated program is then compiled using conventional C++ software tools. C+- is currently supported on computer systems that run the Cosmic Environment [39], which includes multicomputers and networks of workstations. C+- programs can also be executed on the Mosaic C. Section 3.1.4 briefly presents the interface between C+- and the Mosaic runtime system. Chapter 5 details the Mosaic runtime system, which itself is written in C+-.

C+- is a superset of C++, with the exception that global variables are *not* supported in

C+−. C+− extends C++ by including language support for process abstraction and message passing. Advanced features include support for executing computations on heterogeneous ensembles and the capability to send dynamically-defined data structures, *eg*, linked lists, in messages.

3.1.1 Process Abstraction

In C++ and in C+− programmers can define new data types using the **class** construct. These data types can include member variables, member functions and constructors and destructors [37]. Unlike C++ C+− is based on a concurrent model of execution. A mechanism for process abstraction is fundamental for expressing concurrent computations. In C+−, process types can be defined using the keyword **processdef**. Process-type definitions are syntactically analogous to the definition of a C++ class: programmers can define member variables, functions, constructors and destructors. The primary difference between a **processdef** and a **class** is that the process definition implicitly describes a thread of execution that can be concurrent with other processes in the computation.

Program 3.1 contains a naive calculation of the factorial function using C+− processes. The process type **fac** is defined, which contains two local variables, an integer and a pointer to another **fac** process. Two member functions, a **fac** constructor and the **rcv_value** function, define the interface to this process. New C+− processes are created using the **new** operation. Within the **fac** constructor definition, a new **fac** process, with constructor arguments **n-1** and **parent**, is created if **n** is greater than 1.

3.1.2 Message Passing

The public interface to a process consists only of *atomic* functions and constructors. In Program 3.1, a **fac** process communicates with its **parent** process by invoking the **rcv_value** member function (**parent→rcv_value(n)**). *Invoking an atomic function of a process on a set of arguments is equivalent to sending that process a message containing those arguments.* The destination process, upon receiving this message, executes the appropriate atomic function.

Invoking an atomic function that returns **void**, such as the **rcv_value** function, does

```
processdef fac
{   int n;
    fac* parent;

    public:
        fac(int N, fac* P)
        { n = N; parent = P;
          if (n > 1)
             (void) new fac(n-1,this);
          else
             parent→rcv_value(n);
        }

        atomic void rcv_value(int v)
         { parent→rcv_value(v * n); }
};
```

Program 3.1: C+− Program for Computing Factorial.

not suspend the execution of the invoking process. Similarly, invoking the constructor of a process as part of the new operation does not suspend the invoking process if the returning reference value is not used. Void atomic functions and constructors thus provide (potentially) concurrent execution of the invoking and destination processes. Atomic functions can, however, return values, in which case the invoking, or sending, process suspends its execution until the return value is received. *Atomic functions that return values are the C+− syntactic construct for Remote Procedure Calls.*

In addition to supporting RPCs, C+− includes the keywords active and passive so that a process can selectively enable/disable its atomic functions. If a particular atomic function of a process has been made passive, any messages arriving that would invoke that function are queued. This mechanism for selectively receiving messages based on their "type" is significantly less expensive to implement than an RPC. When a process suspends to execute an RPC, the temporary state of that process, *eg*, its stack, must be saved. When the return value is eventually received, this state must be restored before process execution can resume. In contrast, a process is only scheduled to execute by the runtime system if the destination atomic function is active, so the process does not suspend during its execution.

3.1.3 Advanced Features

USER-DEFINED SEND OPERATIONS

Using languages such as C++ and, by extension, C+-, programmers can construct arbitrarily complex data structures. In earlier multicomputer programming systems such as Cosmic C [32], sending these data structures in messages to other processes required that the programmer explicitly "flatten" the structure at runtime. When the message arrives to its destination, that process interprets the "flattened" structure in the message.

In contrast, C+- allows programmers to define special **send** and **recv** member functions in class definitions. In these functions, the programmer defines how the structure should be flattened on the sending side and restored on the receiving side, respectively. This special data handling is invoked at runtime when the data type, or any data type that includes it, is sent in a message.

EXECUTION ON HETEROGENEOUS ENSEMBLES

In *heterogeneous* multicomputers ensembles, some nodes may possess special capabilities *eg*, floating-point accelerators or file-system access. The hardware implementation of these special nodes may be fundamentally different from other nodes in the ensemble. For example, the byte order may be reversed, or a different word size may used. Using C+-, a programmer can express *at compile-time* the characteristics of the different machine types in the ensemble. The programmer defines a special function called **translate** that, given a reference value, returns the machine type of the node on which that process resides. If that machine type is different from the sending node, the contents of the message are translated so that they are consistent with the machine characteristics of the destination node.

This capability is also useful for multicomputer ensembles such as the Mosaic C used for the experiments in this thesis. This ensemble is configured as a set of identical Mosaic nodes that is connected to a host computer. The host "node" is a SUN SPARCstation which has fundamentally different machine characteristics than the nodes internal to the ensemble. The prototype Mosaic runtime system includes a C+- program that executes on the host to provide services such as error reporting and I/O.

In systems such as the Cosmic Environment [39], programmers are required to include code that invokes a special function that translates the contents of the message *each time* that message travels to or from the host. C+- provides an abstraction mechanism that allows the programmer to declare the the characteristics of the target ensemble at compile-time. Any translation required when messages travel between incompatible nodes is handled at runtime.

3.1.4 Runtime System Interface

As mentioned in section 1.3.3, the code for a fine-grain program is often too large to fit entirely on a fine-grain node. Even if the code could fit, dedicating such a large fraction of the total memory of the ensemble to storing thousands of copies of identical code is inefficient. The C+- programming system includes a software linking tool that "splits" the user code into *code pieces.* Each code piece includes the code for one atomic member function plus code for each ordinary function called by that atomic function. The linking tool assigns an index to each code piece and links the program by replacing all program references to the code piece with the index. In sections 5.3 and 5.2.3, we briefly present algorithms for the assignment of code pieces to nodes and algorithms for accessing the code pieces during program execution, respectively.

Execution of the user computation begins with the instantiation of special **root** process. Each C+- user program must include a constructor, called **root::root**, for this process. The argument to the **root** constructor is a class ARGS that contains the **argc** and **argv** command-line arguments. The **root::root** routine is thus the analog of the C++ **main** routine. In all non-trivial computations, the root process initiates the dynamic creation of the set of processes needed for the computation.

When executing on a multicomputer, the **new** of a C+- process (see Program 3.1)

```
class _NODE_;                       // node identifier class

processdef Console                  // definition of console process
{    public:
         atomic void print(char *,...);
};

class IFC
{
   public:
         Console                    *console;

         int*                       malloc(int size);
         void                       free(int* ptr);
         int                        random_num();
         void                       wait();
         _NODE_                     my_node();
         int                        num_nodes();
         void                       halt(int code);
};
```

Program 3.2: C+− Class Interface between C+− Programs and the Mosaic Runtime System.

involves bundling up the information about what type of process is to be created and the constructor arguments within a **new** message. This message is sent to a special **Constructor** process located on a selected node in the ensemble. A node is selected for process placement by calling a PARENT function that is defined as part of the runtime system. This function may use a variety of process-placement algorithms (section 6.2). A **Constructor** process, upon receiving a **new** message, executes a function that allocates memory for the new process. The **Constructor** process then executes the constructor for the new process. When that function finishes executing, the **Constructor** process returns the reference value of the new process to its parent process.

A C+− program interfaces with the Mosaic runtime system via a pointer to an instance of a special IFC class. Program 3.2 details the variables and functions that are accessible to a C+− program.

3.2 Fine-Grain Programming Methods

As experience grows with a new programming model, programming methods emerge that are advantageous to program expression or performance. These methods are frequently suggested by characteristics of the target architecture or by features of available software tools. For example, in sequential programming, the array data structure is efficient because its structure and function is directly supported by the structure and use of the memory in a sequential computer. In compiled code, intermediate values are often stored in registers because register allocation and access is significantly less expensive than a main-memory access.

The existing literature concerning medium-grain multicomputer programming (*eg*, [18]) contains many programmer observations about the techniques used to achieve high performance. For example, since memory is plentiful on a medium-grain node and message passing is relatively expensive, programmers often build and maintain copies of data structures such as look-up tables in every process requiring access. Programmers have also given great care to balancing the process load on the nodes. This load-balancing effort is motivated by the fact that there are not many processors (prompting concern about processor utilization) and that execution times and resource demands of individual processes may vary widely.

Fine-grain characteristics such as small node-memory size, fast message passing, and inexpensive process creation stimulate development of programming methods that are significantly different from those developed for medium-grain multicomputers. In general, *fine-grain programmers should express the maximal concurrency in their application using fine-grain, reactive processes.* The Cantor project, as a experiment in how to write fine-grain programs, was the major vehicle for the development of these methods [4, 8].

3.2.1 Maximal Concurrency

Since a fine-grain multicomputer has a large number of nodes, each of which can support multiple resident processes, an application program may employ an enormous number of concurrent processes. The more processes an application can legitimately use, the more opportunity there is for concurrency. Concurrency is useful provided that there are enough processing resources to exploit it, and that the overhead necessary to achieve concurrency does not outweigh its benefit.

For a fine-grain machine, the general assumption is that there are enough processors to exploit concurrency and the cost of creating a new process is low, roughly the same as the cost of sending a message.[1] For a medium-grain machine, the small number of processors and the overhead of creating processes means that processes should be created sparingly. In the literature, one sees reference to "building up the granularity" of processes in an application. This technique testifies to the fact that high-performance programs must balance the cost of creating a process to execute concurrently versus the amount of work that will be performed by that process. For a medium-grain machine, a process needs to perform many thousands of instructions to overcome the cost of the process's creation. For a fine-grain machine, only a few tens of instructions outweigh the low cost of process creation. Consequently, the rule of thumb in fine-grain programming is: *if operations can be concurrent, spawn new processes to execute them.*

The combination of multiprogramming on each node and the large number of small processes that comprise a computation simplifies the task of distributing the processes across the nodes of the ensemble. In [6], Bhatt presents a load-balancing algorithm that uses a simple randomized algorithm to dynamically embed binary trees into hypercubes. This algorithm requires only local information and has provable bounds on the distance between the parent and the child processes and on the number of processes placed on each node. Our experimental results with embedding arbitrary process graphs into meshes (section 6.2) indicate that similar randomized algorithms perform well for automatic process placement.

Expressing the maximal amount of concurrency in an application from the beginning has another benefit. If the application is to be run on a computer with different characteristics, such as a medium-grain machine, the concurrent elements can be aggregated

[1] The cost of a message-passing operation is roughly equivalent to the time required for a procedure call.

to "build up the granularity." Contrast aggregation of concurrency with the difficulty of extracting parallelism from programs [15]. By avoiding unnecessary synchronizations in the application program, fine-grain programming techniques provide greater flexibility in program execution.

3.2.2 Fine Granularity

A second fine-grain programming method is a direct consequence of the effort to express the maximal concurrency in an application: fine-grain processes are less complex than processes of larger granularity. For a given application, if the computation is divided between more and more processes, in general the amount of work done by each process decreases. The amount of code needed by the process also decreases. Since less work is being done by the individual process, it is likely that the process will operate on a smaller set of private data variables. Processes will depend more heavily on frequent message passing not only to obtain new tasks but also to synchronize with other processes. Since these messages are frequent, they will probably be small, *ie,* processes do not spend significant amounts of time computing and constructing large messages.

Processes require both physical space (memory) and processor resources to execute. Splitting the application into many, small processes can make the runtime system's task of placing those processes on nodes simpler. The memory requirements of the set of processes can be more evenly spread across the nodes if there are many small processes rather than a few, large ones. The analogy is that the fine-grain runtime system can pour fine-grain processes onto the fine-grain machine like sand, filling evenly the overall machine. Balancing the processing load across the nodes of the ensemble is also facilitated by splitting the computation into many pieces that execute for a short period of time[29, page 20].

3.2.3 Reactive Behavior

As discussed in section 1.3.3, the reactive behavior of processes is a characteristic that is ideally suited to fine-grain machines. Since reactive processes consume messages in the order in which they are received, communication deadlock is avoided and the small local memory of the node does not overflow.

In the Cantor experiment [4], all programs were originally written with purely reactive semantics. This programming constraint was sometimes cumbersome, particularly if the algorithm being implemented included use of "conventional" data structures, such as stacks and queues (section 1.2). In these instances, the additional effort required for programming with purely reactive semantics motivated us to consider the addition of a selective-receive mechanism. The queue-problem solution presented in section 1.2.2 illustrates how selective receive can be implemented using purely reactive semantics. Requiring programmers to express selective receive at the level of the queue-problem solution however is unnecessary. By including selective receive in programming notations and runtime system support, programs are simplified and runtime execution of selective receive can be optimized.

The question remains, however, to what extent should selective receive be used? In C+−, programmers use the **active** and **passive** construct to indicate which messages should be consumed next by individual processes. Compatible incoming messages are identified by the runtime system. Even in this simple case, the use of selective receive should be limited as the runtime system must do extra work to alter the order of message consumption.

The selective-receive mechanism can, however, be used to implement a more powerful construct, the remote procedure call (RPC). If a process executes an RPC, it sends out a message, *suspends its own execution*, and then waits to receive a returning RPC-reply message. In a simple selective receive, there is no executing state of the process that must be saved and then later restored when the selective receive completes. For the RPC, the executing state of the process – its stack and all the live register variables – must be preserved so that the process can transparently resume when the RPC completes. The benefit and cost of the basic selective-receive mechanism seem to justify its inclusion in programming notations, while the wisdom of including RPC capabilities remains a research question.

4 Fine-Grain Runtime Systems

The evolution of the multicomputer architecture toward finer granularity has been motivated primarily by the promise of improved cost/performance (section 1.3). To exploit the available concurrency, our multicomputer programming has necessarily evolved toward a fine-grain model (Chapter 3). Since the runtime system is the interface between the machine and its programming, fundamental changes in the physical machine or in the programming are reflected in runtime system design and implementation.

In this chapter, we first describe criteria for fine-grain runtime system design. These criteria are motivated both by the requirements of user programs and by the physical realities of fine-grain hardware. We present a method for designing and constructing fine-grain runtime systems. A complete runtime system for Mosaic ensembles called MADRE (presented in Chapter 5) has been developed using this method. Just as we evaluate our programming ideas by writing and evaluating programs, we investigate the effectiveness of fine-grain runtime system ideas by constructing runtime systems and evaluating their performance through experiment (Chapter 6).

4.1 Runtime System Design Criteria

First of all, we use the term *runtime system* rather than *operating system* to describe the software interface between the Mosaic hardware and user programs. The motivation for this distinction is the connotation that an "operating system" is a software system that handles not only process management, but also support for files and users. While these capabilities are arguably desirable for fine-grain multicomputers, the current effort is to design and implement runtime support that efficiently handles the most critical operations. The term "runtime system" is used here to refer to a simple operating system that handles only the essential elements of runtime support. Historically, a runtime system is an operating system so simple that it is often compiled and loaded together with the application program. The MADRE runtime system discussed in the next chapter is, in fact, compiled and loaded separately from the application program, but still provides only essential support for programs. The modular design of this fine-grain runtime system does permit capabilities such as multi-user environments, file management, or security measures to be integrated into the runtime system, thus turning the "runtime system" into a real operating system.

Many of the design criteria for fine-grain runtime systems were actually design criteria of operating systems of medium-grain machines. However, in some cases, medium-grain machine characteristics, particularly the expense of process creation and message passing, precluded development of distributed algorithms to achieve design goals. In other cases, development of solutions to more critical problems preempted interest in solutions to general problems, leaving so-called "skeletons in the closet." With the advent of working fine-grain hardware, achieving these design goals becomes not only feasible, but critical to the success of the architecture.

4.1.1 Distributed Runtime System

One of the major weaknesses of medium-grain operating systems is their "node-bound" view of the machine. A medium-grain operating system is typically composed of *component* operating systems that run on each node (*eg*, [33]). These component operating systems manage the local resources of the node — performing memory allocation, process scheduling, etc. Global management of the processors and memory, via process placement and load-balancing techniques, is left to the programmer. Since message passing is expensive and each node resembles a conventional workstation, operating-system components usually do not cooperate to perform any global resource management.

For fine-grain machines, the runtime system controls both local and global resource allocation (via process placement). The line between the two types of allocation blurs when one considers that the memory resources on each node are so limited that they can be easily exhausted regardless of the sophistication of the local-memory allocation algorithm.

If a node cannot allocate memory for incoming messages, the computation cannot proceed (see next section). *If resources external to a node can be accessed, the potential exists for improved system reliability, performance, and overall resource utilization.* A fine-grain runtime system thus is composed of components running on each node that communicate and cooperate to manage their collective resources (Figure 4.1).

This definition of a fine-grain runtime system classifies it as a *distributed* runtime system. Traditionally, a *distributed operating system* [17] is an operating system in which global-resource management is performed, using only local state information, in a distributed network of computers. A distributed operating system provides the abstraction of a virtual uniprocessor to the user. The goal of such a system is to give the user program transparent, *sequential* access to global resources. In contrast, a user's fine-grain program requires transparent, *concurrent* access to global resources. The abstraction provided by the distributed fine-grain runtime system is that of a maximally concurrent multicomputer, where each user process can be mapped to a processing node. Using this abstraction, user programs can express the concurrency in the application problem, independent of the number of nodes in the target machine. Multiprogramming on each node, which is critical to hiding message latency (section 2.1.2), supports this abstraction.

In the fine-grain runtime system, each component running on a node can be represented as a single process. Distributed algorithms executed by the runtime system then are implemented using message passing between these component processes (Figure 4.1). This process abstraction permits an intriguing generalization: instead of simply viewing each component as a process, the component itself can be implemented as a collection of fine-grain processes (Figure 4.2). Since the physical placement of these processes is not relevant within the programming model (Chapter 3), we can place these runtime system processes so that each node contains a set of *kernel* processes. Some processes that help manage the resources of an individual node, called *remote* processes, may reside on other physical nodes. By dividing each component runtime system into processes that are spread across the nodes of an ensemble, we express a fine-grain runtime system as a fine-grain

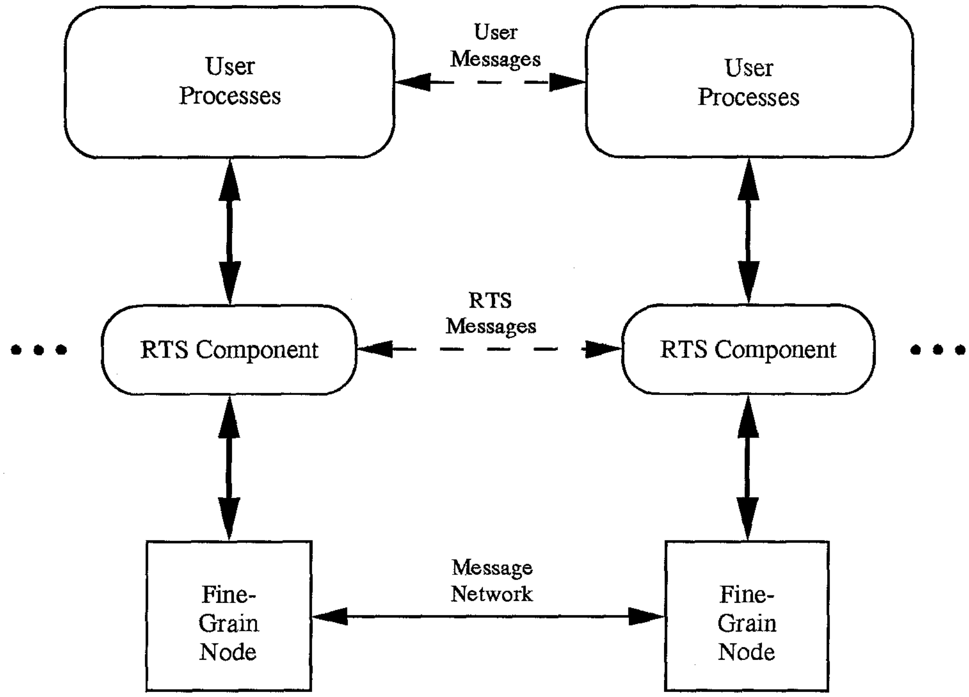


Figure 4.1: Distributed Runtime System Conceptual Organization. The runtime system (RTS) for an ensemble of nodes is partitioned into *component* runtime systems. These component runtime systems provide the interface between the hardware level and the user computation level. Conceptually, messages are sent between parallel levels, *eg*, user processes send messages to other user processes. In practice, outgoing messages travel through levels down to the hardware, across the message network, and then up through the levels on the destination node.

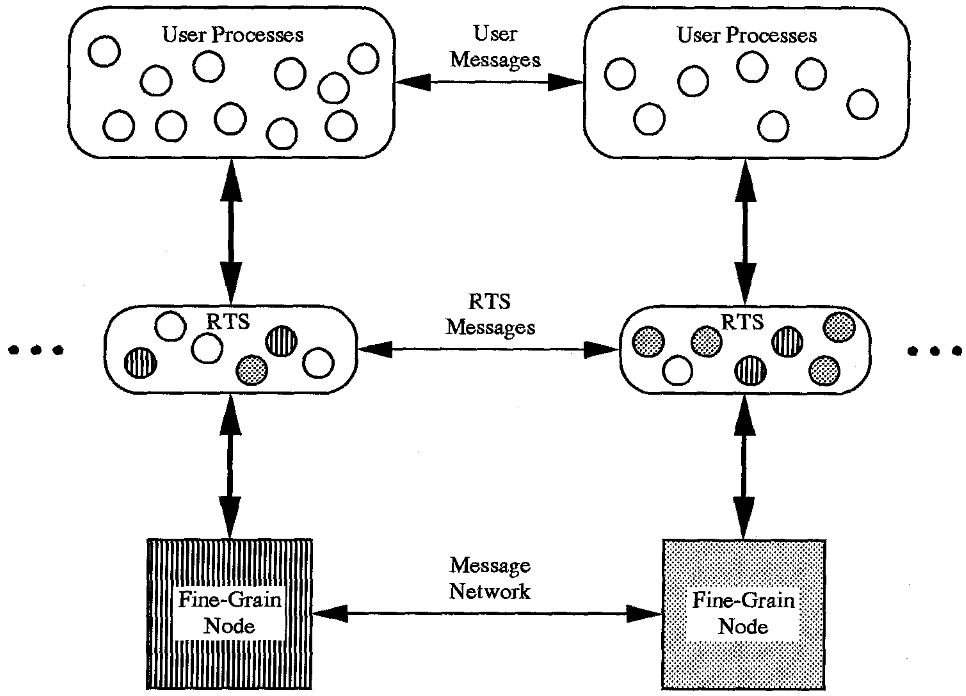


Figure 4.2: Generalization of Components of a Fine-Grain Runtime System. The component runtime system for each node is implemented as a collection of processes. In the figure, runtime system processes are shaded to match the node whose resources they help manage. Some of these processes, called *kernel* processes are physically resident on the node, while others, called *remote* processes, are resident on other nodes.

program. Thus, a fine-grain runtime system is both a distributed runtime system in that it manages distributed resources using local state information, and it is a distributed fine-grain program.

4.1.2 Robust Operation

As described in [24], the assumption that incoming messages will eventually be consumed by the node is fundamental to ensuring that the message network will not deadlock. In general, messages stream into nodes, with the node runtime system consuming the message at the front of its internal receive queue by delivering it to the destination process. If the receive queue becomes full, messages that cannot enter the node remain queued in the communication network. However, if the message at the front of the receive queue is eventually consumed, blocked messages will eventually be consumed, and, thus, the network does not remain blocked indefinitely.

If the message at the front of the queue is not actually consumed, but rather is buffered indefinitely, the consumption assumption can be violated. This problem can occur if messages are buffered, using selective receive for example, and there are no runtime mechanisms to make space available in the receive queue. Medium-grain operating systems provide very few such *robustness* mechanisms to distribute resource demands. Consequently, if the receive queue overflows, the computation cannot proceed. To eliminate the overflow problem during the next run of the computation, the programmer is encouraged to change the process placement and/or message traffic pattern.

Robustness mechanisms are, however, critical for fine-grain machines. Not only are the process placement and message traffic patterns removed from the control of the programmer, but, since the incoming message queue is physically small, the likelihood of queue overflow is significantly increased. The robustness design goal stipulates that *a fine-grain computation should not fail due to a node resource "hot-spot" if the required resource is available elsewhere in the ensemble.* In many application programs, the load on the machine may peak for short periods. If the runtime system can provide robust operation, the computation can survive temporary resource hot-spots and execute to completion.

As the total machine becomes increasingly loaded, it is reasonable to expect that remote resources will be increasingly difficult to obtain. An additional design goal is that the runtime system be capable of detecting that the entire machine is heavily loaded, and then resort to requesting resource from an off-ensemble entity such as a host computer or a disk.

4.1.3 Efficiency

The danger of innovative runtime system design is that the runtime system itself becomes the primary focus. Regardless of the elegance (or lack thereof) of runtime system design, the speed of execution of a user program is the fundamental criterion by which runtime systems should be judged. *The runtime system must strive to execute user programs as efficiently as possible.*

Runtime efforts to improve robustness are a good example of the trade-offs between functionality and efficiency. It is not acceptable to penalize programs running on a lightly loaded machine in order to support mechanisms that will be required *if* the machine becomes heavily loaded. The goal for fine-grain runtime systems is that, while the machine is lightly loaded, the overhead incurred to support robust operation will be minimal. When the machine becomes heavily loaded (when without robust mechanisms, the computation would fail), increased overhead for process management and message handling is more acceptable.

TIME EFFICIENCY

While all runtime systems seek to minimize the amount of extra work needed to execute a computation, the fine-grain runtime system must be particularly concerned with minimizing the overhead for handling processes and messages. The fine-grain programming model (section 1.3.2) is based on the tenet that sending a message or creating a process is roughly equivalent in cost to a procedure call. Since message passing and process creation are frequent operations, any unnecessary overhead can cause significant performance degradation.

One observation about overhead incurred by operating systems for medium-grain multicomputers has had a significant impact on fine-grain runtime system design: many include entirely too much copying. Not only are data values necessarily copied across the nodes of an ensemble during message passing, but data values are copied *within* individual nodes. This second form of copying is not only unnecessary, but it can also be expensive. The copying of each data value requires one memory read operation *and* one memory write operation. If the data values are written initially in the proper section of memory, the vacuous operation of copying within a node can be eliminated. While some instances of copying remain in our prototype runtime systems, implementing a "copy-less" runtime system has been a major design goal in this thesis experiment.

SPACE EFFICIENCY

The efficiency of the runtime system must also be evaluated on a "space" level. Some medium-grain operating systems maintain data structures that contain state information about other nodes in the ensemble. For example, in the Reactive Kernel [33], each node keeps a table containing the relative coordinates of each node in ensemble. Use of this table decreases the time required for each message send, at the cost of dedicating enough memory to represent the entire ensemble. For a Symult S2010 ensemble [33, 31], this table requires a few hundred words. For a Mosaic ensemble, such a table could require 16K words, or *half the memory on the node.*

In addition to avoiding large system data structures, the resident runtime system on each node must be quite small, ideally a few kilobytes. Adding sophisticated algorithms to the runtime system can significantly increase the physical size of the runtime system code. Every word of memory consumed by the runtime system decreases the amount of memory available to the application program.

4.1.4 Modularity and Extensibility

Modularity and extensibility are listed in the design goals of all operating systems: an operating system should not need major modification whenever a new peripheral is added.

Analogously, fine-grain runtime systems should not need to be re-tooled if an additional thousand nodes are added to the ensemble.

This design goal extends further for fine-grain machines. The fine-grain multicomputer architecture is itself so flexible that a *practical* ensemble can be as many as tens of thousands of nodes or as few as one node. The range of applications for ensembles is equally broad. Consider a small embedded application program with static process placement and message traffic patterns, running on a 4-node ensemble. The runtime system that executes this program may not need to include sophisticated robustness capabilities; the work and space that would be required for such algorithms could be better used for directly running the application. Conversely, a large long-running application program that exhibits very dynamic process placement and message traffic, running on a 16K-node ensemble, may require all the robustness a runtime system can provide. *The extensibility goal for fine-grain runtime systems is then to design runtime systems whose capabilities are matched both to the machine ensemble and to the range of application programs that will be executed.* Modular design of runtime system components, so that capabilities can be added and removed without affecting the overall structure of the system, is the best way to provide this flexibility.

4.2 Runtime System Design Method

4.2.1 Process Layering

As mentioned in section 4.1.1, the distributed runtime system for a fine-grain machine can be implemented as collection of processes. Rather than the runtime system being a lump of code that exists to bridge a gap between machine and programming system, *a fine-grain runtime system is itself a fine-grain program running on a fine-grain machine.* Conceptually, a user computation is a collection of processes that is managed by runtime system processes. Since a process is an abstraction of a multicomputer node (section 1.3.2), the hardware level can also be modeled as a collection of processes. The processes in each level can be organized into one or more *layers*. In Figure 4.3, processes in the runtime-system and user level are

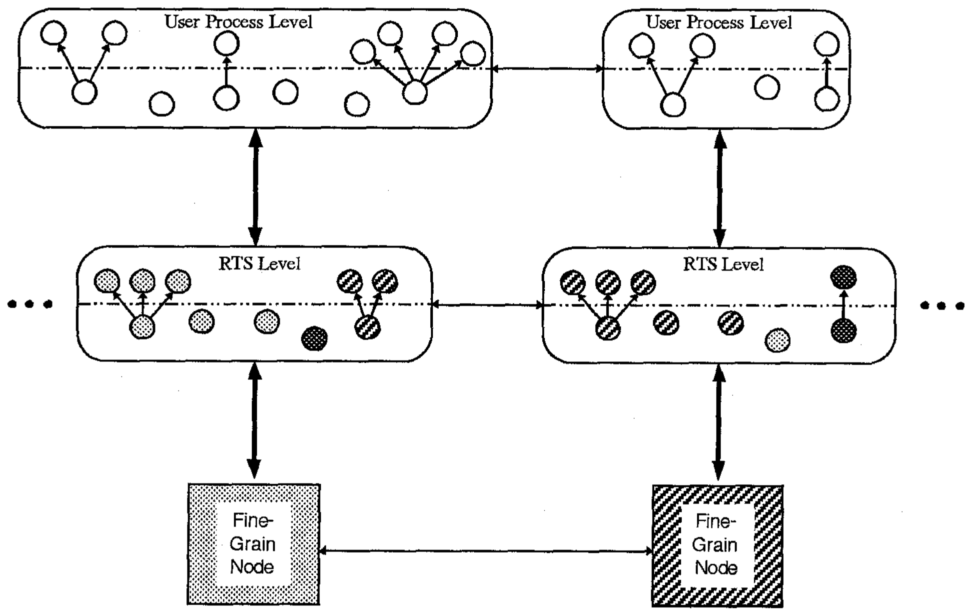


Figure 4.3: Layered Organization of a Distributed Runtime System. As illustrated in Figure 4.2, each component of the runtime system is implemented as a collection of processes, some of which may be resident on other nodes (the shading of a process indicates the node with which it is associated.) Processes may be *layered* within a level. For example, a runtime-system kernel process may dispatch to another kernel process. A kernel process may in turn dispatch to a user-level process.

organized into two layers.

The structure of messages reflects the process layering (Figure 4.4). For example, a user message (*ie*, message sent from one user process to another) will include information to route the message to the correct destination node (the Δx, Δy relative coordinates), through one (or more) destination runtime-system processes and eventually to the destination user process. As the message percolates through each process layer, the message segment pertaining to that layer is interpreted and conceptually removed from the front of the message. The runtime system in effect interprets and "executes" the message as if it were a sequence of instructions.

The layering of a user computation atop a runtime or operating system is commonplace. However, the abstraction of layered processes and the mirroring structure of messages suggests a unified representation for all levels of computation. Such a unified abstraction for process layers and the interfaces between them is a powerful organizational tool. The MADRE runtime system (discussed in chapter 5) illustrates how this abstraction can be used to organize runtime systems. A practical benefit of this abstraction is that exposing structure information to a compiler can simplify runtime support through optimization and error-checking.

An immediate observation is that process layering is not easily implemented using the process model outlined in section 1.3.2. That model maintains that processes are invoked only in response to the arrival of a message. How does a process in one layer *dispatch*, *ie* pass control and a message, to a process in another layer? Explicitly sending the message to the next process layer means in the general case that the message enters the message network, re-enters the node, and then again needs to be dispatched upon. Equally fundamental is the question of data communication between the layers of processes. Recall that, in our model, each process executes in a *private* address space. If each process executes in a totally private address space, how can user processes access low-level parameters such as **mynode_id**? Clearly, each process should not maintain identical copies of such data nor should message passing be required to access the data.

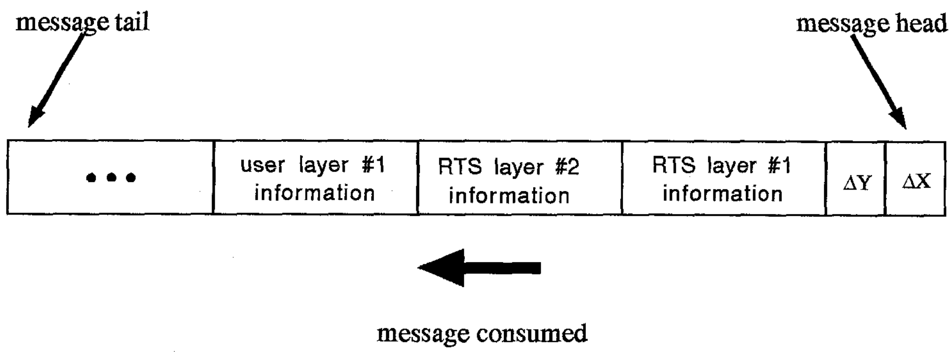


Figure 4.4: Message Layering. The structure of messages reflects the process layering. The message in this figure reflects the process layering in Figure 4.3. A user message (*ie*, message sent from one user process to another) will include information to route the message to the correct destination node (the ΔX, ΔY relative coordinates), through one (or more) destination runtime-system processes, and eventually to the destination user process. As the message percolates through each process layer, the message segment pertaining to that layer is interpreted and conceptually removed from the front of the message.

```
typedef struct { FUNC_PTR entrypoint; DATA_PTR data;}
PROCESS;
...
MESSAGE *msg;
PROCESS *process;
...
/* Decode message header. */
process = lookup_destination_process(msg);
/* Dispatch to embedded process. */
(*process→entrypoint)(process,msg);
...
```

Program 4.1: Runtime Dispatch to an Embedded Process. An embedded process is represented by a pointer to the code to be executed in response to a message, and a pointer to the private data of the process. An embedded process is invoked by dereferencing the function pointer; the arguments passed are the pointer to its data and a pointer to the message to be consumed. (Adapted from [30])

The proposed process-layering capability is summarized as follows:

- Processes can be layered; the structure of messages sent to processes mirrors the process-layering structure.
- Processes can be provided with access to data and system services from processes *in the previous layer.*
- Processes can decode a message section and then dispatch to a process in the next layer.

IMPLEMENTATION

Embedded Processes The questions of process dispatch and data communication have been addressed previously in multicomputer runtime support systems. In [30, page 161], Seitz describes a dispatch mechanism, used at the runtime-system level, that assumes user processes are implemented as *embedded* processes (Program 4.1). An embedded process is represented by a pointer to the code to be executed in response to a message, and a pointer to the private data of the process. In this case, a process invokes an embedded process by dereferencing the function pointer. The arguments passed to the embedded process are the pointer to its data and a pointer to the message to be consumed. In the systems described in [30], there is however no unified mechanism for providing specifying which data and services should be provided to individual processes. All functions are universally available to processes.

C+− The development of C+− provides an opportunity to support process layering directly in user programs. The inclusion of explicit process layering was motivated by the implementation of MADRE, the Mosaic runtime system, in C+−. Since each layer of a fine-grain computation from the user level down to the hardware level can be viewed as a collection of processes, the entire computation should be expressible in C+−. The basic idea for C+− process layering is that processes, when created, can be *dynamically* derived from processes that have already been instantiated[1]. By using the member-protection capabilities of C++ and C+−, dynamic process inheritance can provide controlled access to data and services in other process layers. Currently, work is proceeding on integrating mechanisms for process inheritance, process dispatch and for associating the structure of messages with process layers [34].

The structure of messages should reflect the structure of process derivation. A message sent to a derived process type includes segments that are interpreted by base processes. In C+−, each process type must include, or inherit from a base process, a HEAD and a TAIL function. The C+− translator translates atomic-function invocations into message-passing

[1] In the remaining text, the term *derived* process is equivalent to *dynamically derived* process.

operations by first generating a call to the HEAD function for the destination process type. This function allocates a message buffer, initializes the header of that buffer, and returns a pointer to segment of the buffer where the message arguments should be written. After the message arguments are written into the buffer, the TAIL function is called, which should in turn invoke a function to enqueue the message for sending.

When C+- fully supports process derivation, the generation of message segments pertaining to each process layer will be statically defined by the process-type declaration. In the prototype MADRE system, however, the HEAD and TAIL functions are used to assemble explicitly the outgoing message segments (section 5.2.2).

4.2.2 Runtime-System Level

As illustrated in Figure 4.1, the resources of each node in the ensemble are managed by a *component* of the ensemble's runtime system. For truly distributed runtime systems (section 4.1.1 and Figure 4.3), each component of the runtime system is decomposed into *kernel* processes, processes resident on the physical node, and *remote* processes, which may reside on other nodes.

Kernel Processes

Kernel processes, by being resident on a node, can directly manage node resources. A kernel process is used to encapsulate a runtime system capability that will be executed frequently or with very low latency. For example, node-memory management should be controlled by a kernel process. If the process that manages node-memory allocation were physically located on another node, efficiency would be sacrificed for little or no benefit.

Each component runtime system is expressed as a small program in which kernel processes are statically instantiated. This component program is the program loaded onto each node when the ensemble is reset (section 2.1.3). Program 4.2 illustrates how a particular set of kernel processes for a component runtime system is defined and instantiated using C+-. This component runtime system contains three kernel processes. When this program begins executing, kernel processes A, B0, and B1 are allocated and initialized automatically.

```
processdef Kernel_process_A
{ ...}
processdef Kernel_process_B
{ ...}

Kernel_process_A A;
Kernel_process_B B0,B1;
...
main()
{ ...}
```

Program 4.2: Instantiation of Kernel Processes. Each component runtime system is expressed as a small program in which kernel processes are statically instantiated. This component program is the program loaded onto each node when the ensemble is reset (section 2.1.3). When this program begins executing, kernel processes A, B0, and B1 are allocated and initialized automatically.

Static instantiation has several benefits. First, by compiling the kernel processes as part of the component runtime system, and then placing a component on each node, we control kernel process placement. The desired set of kernel processes will reside on every node. Second, compiling kernel processes together provides the opportunity to streamline the runtime system. Ordinarily, kernel processes on the same node would communicate by message passing. This message passing can be eliminated by having the kernel processes directly invoke the atomic functions of other kernel processes. Third, using C+-, the set of kernel processes is instantiated when the component runtime-system program begins executing. In previous operating systems, such as the Reactive Kernel [33], substantial effort was required to create a process (the spawn handler, in that case) that would be used to create additional processes.

Kernel processes may be layered, so that kernel processes can inherit access to certain data and functions. Given that process layering is currently not provided in C+-, MADRE, a fine-grain runtime system developed for the Mosaic C, implements the desired inheritance of data and services by declaring that all kernel types are friends. By declaring two process types X and Y as friends, all functions in processes of type X can access the private and

```
processdef Remote_process_A
{...}

f()
{...
  Remote_process_A *R = new Remote_process_A (args);
  ...
}
```

Program 4.3: Remote Process Instantiation. Component runtime systems and kernel processes may dynamically instantiate *remote* processes on other nodes. A remote process is the primary mechanism by which component runtime systems can use non-local resources to satisfy local resource demands.

protected parts of processes of type Y. This "implementation" of process layering permits the inheritance of data and functions across process layers, but at the cost of removing any protection features that are desired. For the user processes, special locations in memory are loaded with pointers to a few runtime system processes. Using these pointers, user processes can access public data and functions in the runtime-system kernel processes.

Whether the functions provided by kernel processes include mechanisms for selective receive and/or Remote Procedure Call is an important design decision. Adjacent layers of processes must cooperate to provide a selective receive or an RPC. Ordinarily, messages are consumed as they percolate through the layers of processes. When a selective receive or RPC is performed, a lower layer of processes buffers some messages rather than dispatching them to the next layer of processes. Thus, unless the fine-grain hardware includes message buffering capabilities, the lowest layer of kernel processes cannot buffer messages. Subsequent layers of kernel processes can rely on buffering mechanisms only if provided by a lower layer of processes.

Remote Processes

Component runtime systems and kernel processes may dynamically instantiate runtime system processes called *remote* processes on remote nodes (Program 4.3). Remote-process creation is the primary mechanism by which component runtime systems can use non-local resources to satisfy local resource demands. Using the remote system process mechanism, data structures relevant to the resource management of a node can be distributed across many nodes in the ensemble (Figure 4.3).

The number and types of remote processes used by individual component runtime systems may vary. *Just as process creation is the mechanism of growth in user computations, remote processes are the mechanism by which the component runtime system of a node can grow.* Where necessary, remote processes may be layered, as for kernel processes.

When a remote process is instantiated on a remote node, we are, in effect, allocating remote memory for use by the local node. It is important to note, however, that the remote-system-process mechanism provides more than just memory allocation. Since the remote memory is encapsulated in a process, that process can have atomic and ordinary functions defined for it. For example, by encapsulating a buffered message in a remote process, space can be made available in a node's receive queue so that incoming messages can be received. The remote process that contains the exported message may have atomic functions defined for modifying, retrieving, or deleting the message. The resident part of the component runtime system invokes these remote process atomic functions via message passing, thus providing potential concurrency within the component runtime system itself.

Since remote processes are, by definition, not allocated on the same node as the runtime system process that created them, they do not dynamically inherit data or functions from the kernel process structure. Remote processes may rely on message buffering under the constraints analogous to those for kernel processes: a lower layer of remote processes must provide the buffering capability.

4.2.3 Correctness Requirements

To build a runtime system, or to extend the capabilities of an existing one, additional kernel or remote processes may be defined. For each additional kernel process definition, two conditions must hold. First, the operations on any data inherited from higher levels are safe. Many processes may be derived from a higher level process, so the operation of derived processes must not corrupt shared data. Second, the distributed algorithm executed by the collection of new kernel processes must make progress.

Since remote processes do not inherit data from other layers of computation, the access of data by the remote process is guaranteed to be safe. However, if additional layers of processes are derived from remote processes, access of data within the remote process structure must be safe. The algorithms implemented by the remote process structure must also make progress.

5 MADRE: The Mosaic Runtime System

Chapter 4 describes the design criteria and a strategy for constructing fine-grain runtime systems. Many questions remain however. Can a runtime system for a fine-grain machine be *efficiently* expressed as a collection of fine-grain processes that are distributed across the nodes of the ensemble? Can a fine-grain runtime system deliver the promised concurrency to application programs with reliability and a minimum of overhead? Does the organizational strategy outlined in Chapter 4 allow a fine-grain runtime system to be modified or extended depending upon the machine configuration and the spectrum of application programs?

The MADRE (**M**os**A**ic **D**istributed **R**untim**E**) system is a fine-grain runtime system expressed as a fine-grain program. C+-, with its abstractions for processes and messages, is an ideal notation for expressing the MADRE program. Since the programming and runtime-system levels are distinct, this choice of notation for the runtime system does not necessarily affect the choice of programming notation for user programs. Any programming notation whose base model is processes and messages can be supported directly by the runtime system. Currently, the MADRE system runs atop a Mosaic C node, and supports user programs written in C+-.

The MADRE system, the C+- programming notation, and the Mosaic C multicomputer form the experimental apparatus used to investigate the questions posed above. In this chapter, we present the general organization of the MADRE system, and then a detailed description of individual components. A rudimentary host interface for I/O and loading is also presented.

5.1 Structure

Figure 5.1 illustrates the conceptual organization of the MADRE system. MADRE

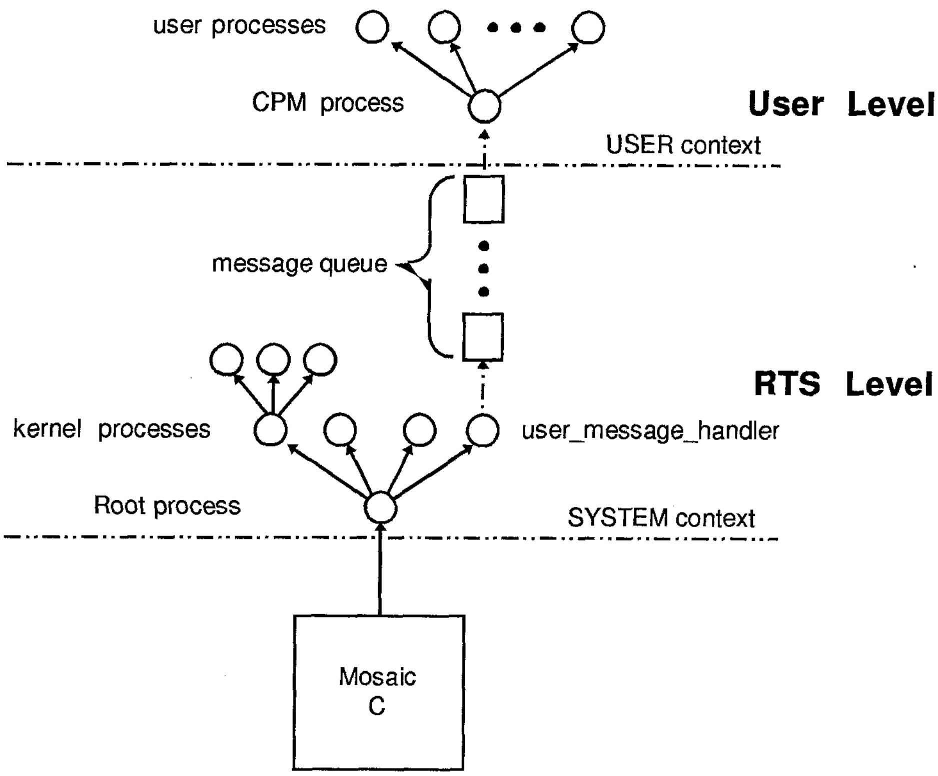


Figure 5.1: Conceptual Structure of the MADRE System. Runtime-system processes execute in the SYSTEM context. When an incoming message has been written into memory, an interrupt is generated by the hardware and the MADRE **Root** process is invoked. The **Root** process services the interrupt by decoding a section of the message, and then dispatching to one of the statically instantiated kernel processes (solid arrows represent dispatch). One kernel process, the **user_message_handler**, queues user messages for consumption by processes executing in the USER context. The **CPM** process consumes this queue by decoding the messages and dispatching to C+− user processes.

processes are organized in layers as described in section 4.2.1. A process operates on a section of an incoming message and may then dispatch to a process in another layer. Using C+- terminology, dispatch is equivalent to invoking an atomic function of a process in another layer. Since C+- does not currently provide direct support for process layering, the MADRE system implements process inheritance explicitly (see section 4.2.2).

The lowest layer of the MADRE system contains a single process called the Root. The Root process is defined by a small program that includes interrupt-handling routines and the static instantiation of the set of kernel processes, as illustrated in Program 4.2. The runtime system for the ensemble is loaded by instantiating a Root process on each node of the ensemble.

When an incoming message has been written into memory, an interrupt is generated (section 2.1.5) and the MADRE Root process is invoked. The Root process services the interrupt by decoding a section of the message and then dispatching to one of the statically instantiated kernel processes. This low-level organization is similar to the organization of the Reactive Kernel [33]. The processes to which the Root process dispatches can be viewed as *handlers*. Handlers may in turn dispatch to other MADRE processes. User messages percolate up through layers of the system, and are eventually consumed by user processes.

An incoming message can be categorized by its final destination, which is either a user process or a MADRE process. Messages sent between MADRE processes are usually part of a distributed resource-management algorithm executed by the collective runtime system of the ensemble. Since the efficiency of these algorithms is crucial to performance and/or robustness, these messages must be consumed as quickly as possible. Consequently, MADRE messages are not internally queued; they are processed immediately.

An internal message queue is used to queue incoming user messages for consumption by user processes. This internal queue decouples the arrival of messages from their consumption, thus ensuring that messages are removed from the network as quickly as possible. As illustrated in Figure 5.1, one MADRE handler, called the user_message_handler, is responsible for queuing these user messages. Each user message contains a section that indicates that the message should be routed through this queue-producer handler. After the message has been appended to the user-process message queue, the **Root** process acknowledges, *ie*, clears, the interrupt.

The internal user-message queue is consumed by the CPM (**C** **P**lus **M**inus) process. This process repeatedly executes a loop that removes the message at the front of the queue, decodes the message to identify the destination user process, and dispatches to that process. When the message has been consumed, control returns to the CPM process. *The specification of the CPM process and the* ***user_message_handler*** *primarily defines the support for different programming languages.* If a programming language other than C+– is to be supported, these two processes would need to be re-tooled to accommodate the desired user-message consumption algorithm.

5.1.1 Use of Dual Contexts

The two contexts of the Mosaic node are used to execute the two software levels of computation illustrated in Figure 5.1. MADRE processes execute in the priority SYSTEM context, whereas user processes and the CPM process execute in the non-priority USER context. Since progress is made in the user computation only when user code is executed, the amount of time spent in the SYSTEM context should be minimized.

The normal mode of operation is that the program in the non-priority, USER context is executing – the CPM process is removing messages from the internal queue and dispatching to user processes. When an interrupt arrives, the processor switches to the SYSTEM context to execute code to service the interrupt. After acknowledging the interrupt, the processor resumes processing in the USER context. Thus, the user computation is executed by default, with the runtime-system code being executed in response to interrupts.

5.2 Components

In the following subsections, we detail the component processes in the MADRE system – the hardware node "process", the **Root** process, the various MADRE kernel and remote processes, and the CPM process. For each process, the following information is described:

- the message section interpreted,
- the algorithm executed,
- the pool of processes to which it may dispatch,
- the data and services that it provides to a derived process (*ie*, a process in a subsequent layer).

5.2.1 The Mosaic Node "Process"

As discussed in section 4.2.1, extending the process abstraction down to the hardware provides a unified abstraction for all the layers of a fine-grain system. The hardware layer, the Mosaic-C node, provides the lowest-level data and services. All processes on the node are conceptually derived from the node "process" so they can access these data and services. Data that are available to derived processes include data registers (R0...R15), address registers (MRP, MRL, ...), and each node-memory storage location. Services that are available include message sending (by writing a value in the DXDY register), and acknowledging interrupts (by writing the value of the interrupt back to the Interrupt Status Register).

The Mosaic node is a process that is defined in hardware rather than in software. For software processes, a process is invoked to interpret some section of a message, and then (potentially) dispatch to another process. The analogy for a Mosaic node is that it detects a hardware event *eg*, a message has been received, a message has been sent, or the message receive buffer has been exhausted, as its "message." The MADRE **Root** process is the only process to which the node dispatches.

5.2.2 The Root Process

THE INTERRUPT "MESSAGE"

The "message" decoded by the MADRE **Root** process is the interrupt generated by the Mosaic node hardware. Special address registers MRP (Message Receive Pointer), MRL (Message Receive Limit), MSP (Message Send Pointer), and MSL (Message Send Limit) are used to delimit incoming and outgoing messages, respectively. The three Mosaic interrupts (described in section 2.1.5) are:

- **receive_interrupt** indicating that an entire message has been written into the region of node memory between the MRP and MRL specified by the runtime system,
- **send_interrupt** indicating that an outgoing message, delimited by the values of MSP and MSL specified by the runtime system, has completely entered the message network,
- **buffer_full_interrupt** indicating that the write buffer provided for an incoming message has been exhausted (*ie*, MRP == MRL).

When an interrupt is generated, the **Root** process services and then acknowledges the interrupt.

Two-Phase Receive In section 2.1.5, we outlined a straightforward use of Mosaic interrupts to receive messages. The processor configures MRP and MRL so that incoming messages are written into a region of memory maintained as a circular queue. One disadvantage of this algorithm is that messages that cannot be immediately consumed, *ie* buffered messages, would probably need to be copied from the queue to another region of memory. This approach would violate the "copy-less" runtime-system design goal (section 4.1.3). To avoid copying, the processor could configure MRP and MRL so that incoming messages are written into preallocated message buffers. If the message were longer than the allocated buffer, a **buffer_full_interrupt** would be generated, and a new, larger, buffer would be allocated. The partially-received message would be reconstructed in the new buffer. Using this strategy, the processor would probably allocate blocks that are larger than necessary to avoid the expense of buffer overflow.

The Mosaic interrupt mechanism can be used to allocate a receive buffer of the correct size *before* the message is actually received. Figures 5.2 and 5.3 illustrate the *two-phase receive* protocol. Every message contains a small header that includes:

- the length of the message,

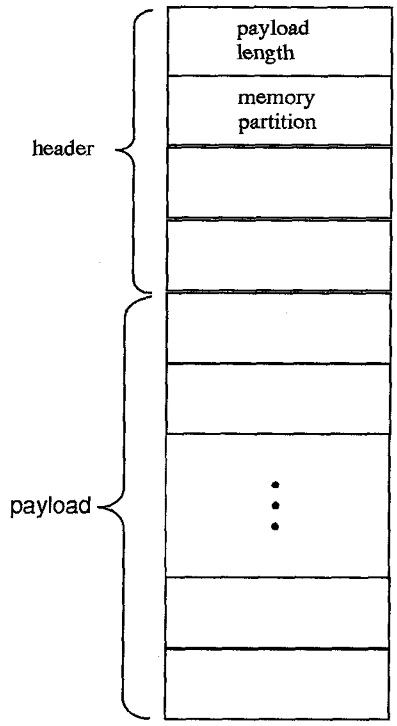


Figure 5.2: Message Structure. Each incoming message contains a header that includes the length of the non-header part of the message (*ie*, the message payload), the partition of memory where the message should be written and a few other words of information. Before the first phase of the receive, MRP and MRL are set to point to the boundaries of the **msg_hdr_buffer**, which has the same structure as the message header. When an incoming message arrives, the header of that message will be written into the **msg_hdr_buffer** until the buffer is full (MRP == MRL). At that point, a **buffer_full_interrupt** is generated.

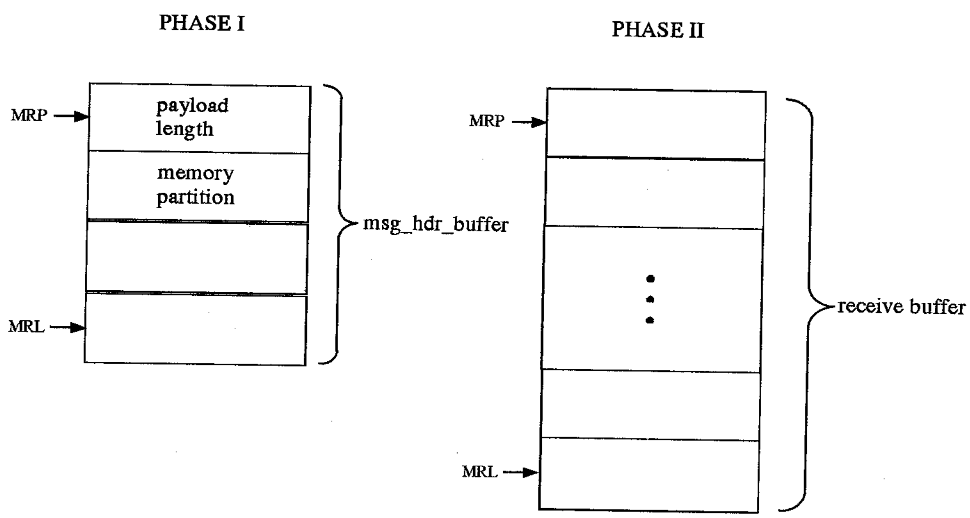


Figure 5.3: Two-Phase Receive Protocol. The interrupt servicing routines of the **Root** process extract the memory partition and the size of the incoming message from **msg_hdr_buffer** and allocate a receive buffer. MRP and MRL are set to point to the boundaries of this new buffer. The second phase of the receive begins when the **buffer_full_interrupt** is cleared. The remainder of the incoming message is written directly into the receive buffer.

- the partition of memory where the message should be written (Figure 5.4),
- a few words of additional information, such as the sending node, the reference value of the destination handler, and the atomic function of that handler that should be invoked.

A special buffer called **msg_hdr_buffer** is allocated statically within the MADRE **Root** process. This buffer has the same structure as the header of each message. Before the first phase of the receive, MRP and MRL are set to point to the boundaries of the **msg_hdr_buffer**. When an incoming message arrives, the header of that message will be written into the **msg_hdr_buffer** until the buffer is full (MRP == MRL). At that point, a **buffer_full_interrupt** is generated. The interrupt-servicing routines of the **Root** process extract the information about the memory partition and the size of the incoming message from **msg_hdr_buffer** and allocate a receive buffer. MRP and MRL are set to point to the boundaries of this new buffer. The second phase of the receive begins when the **buffer_full_interrupt** is cleared. The remainder of the incoming message is written directly into the receive buffer. This two-phase receive can be used to allocate a receive buffer of exactly the correct size without unnecessary copying.

Using this protocol has two notable disadvantages. During the first phase of the receive, incoming messages are blocked back into the network while the **Root** process allocates a message buffer. Heuristics will be used in memory allocation to balance the time required to allocate a new buffer versus the memory utilization. The second point is not as crucial: the header information is not received contiguously with the rest of the message.

The development of the two-phase receive protocol testifies to the remarkable flexibility of the Mosaic interrupt mechanism. This use of the interrupts for receiving messages was developed long after the Mosaic had been designed and prototyped. If the protocols for sending or receiving messages had been built into hardware, we would not be able to experiment with the two-phase-receive protocol.

Process Pool for the Root

In the current MADRE system, a reference value of a process is a 2-tuple

(**NODE**, memory location on the node)

where **NODE** is another 2-tuple (x coordinate, y coordinate). The **msg_hdr_buffer** includes the reference value of destination handler and the atomic function of that handler that should be invoked. The pool of processes to which the MADRE **Root** may dispatch, *ie*, handlers, is defined to be the set of kernel processes (section 5.2.3).

For simplicity and efficiency, the MADRE system exploits "forged" references at the handler level. Ordinarily, processes obtain the reference values of other processes through process creation, or through message passing. Requiring each handler process on every node to explicitly obtain the reference for handler processes on other nodes before computation can begin is inefficient and unnecessary. Recall that the kernel processes are statically instantiated within the **Root** process. In the prototype MADRE system, every node in the ensemble is loaded identically with a **Root** process. Thus, the memory location within a node of a particular kernel process is the same on every node. The reference value of a given handler on a remote node is "forged" by constructing a 2-tuple containing the remote-node identifier and the *local* memory address of that handler.

Root Data

The MADRE **Root** process provides the following data to derived kernel processes:

mynode the node identifier for this node,

xdim, ydim the number of nodes in the x and y dimensions of the ensemble, respectively,

num_nodes the number of nodes in the ensemble,

console_ref the reference to the output device for this node,

error_ref the reference to the error-reporting device for this node,

disk the reference to a disk (*ie*, host) that may be connected to the ensemble.

By invoking atomic functions of the **console_ref**, **error_ref** and **disk** processes, derived processes can access these ensemble resources.

Root Services

The services provided by the MADRE **Root** process to derived processes include:

- **malloc** and **free** for local-memory allocation and deallocation,
- message-sending functions including as **enqueue_msg** for queueing outgoing user messages (section 5.2.2),
- **halt** to halt the node upon an error condition.

Memory Management Each Mosaic node contains 32 KW (16-bit words) of dynamic memory, of which approximately 7–10 KW are consumed by the resident MADRE system. The MADRE **Root** process manages the allocation and deallocation of the remaining local memory via **malloc** and **free** functions that are inherited by kernel processes. (Recall that remote processes do not inherit data or services from the **Root**.) The algorithms used to manage the allocation/deallocation of local memory are generally independent of the rest of the MADRE system. However, as illustrated in section 5.2.3, some kernel processes require knowledge of the memory-allocation strategy.

When the MADRE **Root** process is instantiated on a Mosaic node and begins executing, it segments the available memory into four partitions – the **RCVQ**, **SNDQ**, **SYS** and **PROC**. Figure 5.4 illustrates the conceptual memory map of a Mosaic node. The **RCVQ** partition is the region of memory where blocks are allocated for incoming user messages. The **SNDQ** partition is used for allocating blocks for outgoing user messages. The small **SYS** partition is where incoming and outgoing runtime-system messages are written. The **PROC** partition is the region of memory where all other allocations are performed, *eg*, allocation for incoming runtime-system messages, instantiation of new processes, and allocation of tables used by the runtime system.

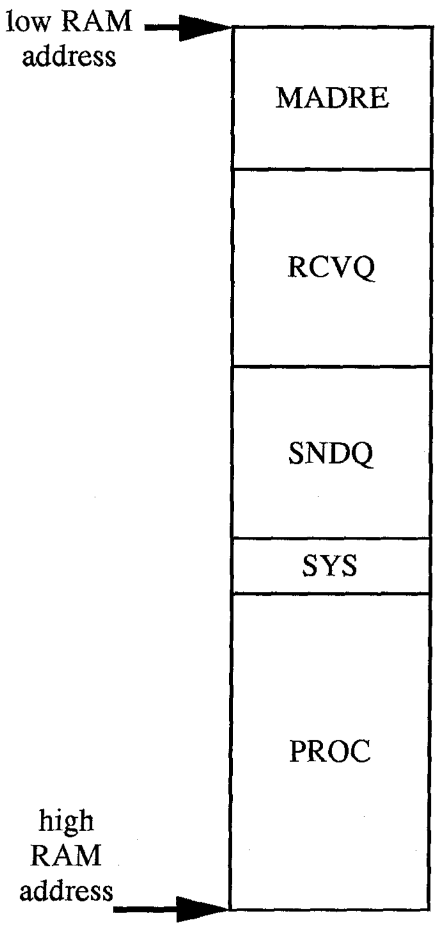


Figure 5.4: Map of Mosaic Node Memory. After the MADRE system is loaded into the memory of each node, the remaining memory is partitioned into four segments – the RCVQ partition, where incoming user messages are written, the SNDQ partition, where outgoing user messages are constructed, the small SYS partition, where incoming and outgoing runtime-system messages are written, and the PROC partition, where all other allocations are made.

The sizes of these partitions are fixed at initialization using heuristic algorithms[1]. The size of the RCVQ partition has significant impact on the overall performance of a computation. If it is too small, processing time must be devoted to ensuring that the consumption assumption is satisfied. If it is too large, space that could be used for instantiating user processes is wasted. In practice, outgoing messages are almost always injected immediately into the message network; hence, the SNDQ partition can be small. The SYS partition must be large enough to ensure that runtime-system messages can always enter or leave the node (section 5.2.3). Since the maximum size of the computation that can be executed on a given ensemble is limited by the number and size of the processes used, the PROC space is the largest partition of memory.

The algorithm used to manage the memory in the SNDQ, RCVQ and PROC partitions is the buddy-system algorithm described in [20]. The available node memory in each partition is divided into blocks according to a sequence of Fibonacci numbers (Figure 5.5). Each block contains header information that includes the block size (equivalently, an index of the Fibonacci number used), the memory partition, and a pointer to the block's "buddy." The free list of blocks is represented by an array of pointers to available blocks (Figure 5.6). Programs 5.1 and 5.2 outline the essential elements of the buddy-system allocation and deallocation routines.

For most memory-allocation strategies, the memory utilization of the algorithm must be balanced against the time required for each resource-management operation. For example, the algorithms that manage memory with little fragmentation or waste are generally slow, while algorithms that allocate or deallocate memory quickly usually reduce memory utilization. The buddy-system algorithms are also sensitive to the trade-off between speed and efficiency. Memory allocation is much faster if a block of the required size is already free; splitting a large block repeatedly can require hundreds of machine instructions. However, leaving free blocks un-coalesced fragments memory and may significantly delay the allocation of large blocks.

[1]Future versions of MADRE will likely include variable partition sizes.

Assume a block of **n** words is requested.

1. The **index** of the lowest Fibonacci number that is greater than **n** is computed and copied into a temporary variable **In**.

2. While the free list corresponding to **In** is empty, increment **In**.

3. If the maximum index of the Fibonacci sequence is reached before a free block is found, no block can currently be allocated so a failure code is returned[a]. If a block is available and that block corresponds to **index**, that block is returned and the allocation is complete.

   Otherwise, while the index **In** of the available block exceeds **index**,

   (a) split the block into blocks corresponding indices **In**−2 and **In**−1,

   (b) append the block corresponding to **In**−2 to the appropriate free list,

   (c) **In** = **In**−1.

   Return block of size **index**.

[a] When allocation for the RCVQ partitions fails, robustness measures are required for the computation to proceed (section 5.2.3). If allocation for the SNDQ partition fails, the processor busy-waits until the send queue empties sufficiently. If the PROC partition is full, the node cannot instantiate any new processes.

Program 5.1: Buddy-System Memory Allocation.

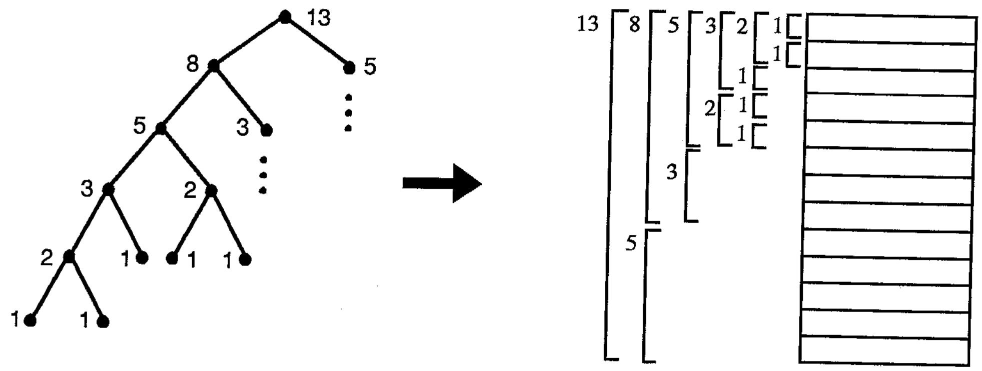


Figure 5.5: A Fibonacci Sequence Used to Partition Memory Blocks. This figure illustrates how a sequence of Fibonacci numbers $(1, 1, 2, 3, 5, 8, 13, \ldots)$ can be used to segment a block of memory. A block of size 13 can be divided into two blocks of sizes 8 and 5, respectively. The size-8 block can then be partitioned into blocks of sizes 5 and 3. Note that a block of size 13, 8, 5, 3, 2, and 1 share the same physical address.

Some simple heuristics are used to the MADRE system to strike a balance between memory-management speed and efficiency. The MADRE execution model assumes that most messages will be small; thus, blocks should not be coalesced beyond a certain threshold of Fibonacci numbers. If the compiler provides information about the sizes of potential messages, the MADRE system can preferentially leave free blocks of those sizes un-coalesced.

Sending A Message The structure of messages reflects the process-layering structure at the destination process that must be traversed. When process layering is fully supported, C+− will automatically generate code that constructs messages whose structure reflects the defined process layering. Currently, layered messages are constructed explicitly using the HEAD and TAIL functions (section 4.2.1) and a class-derivation tree for the message

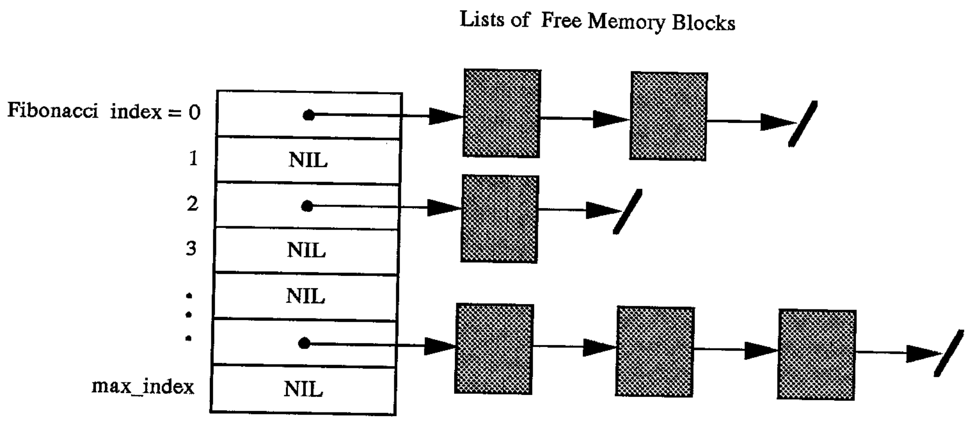


Figure 5.6: Buddy System Data Structures. Each block contains header information that includes the block size (equivalently, an index of the Fibonacci number used) and a pointer to the block's "buddy." The free list of blocks is represented by an array of pointers to available blocks.

segments.

Every message in the MADRE system includes a header that will be written into the msg_hdr_buffer during the first phase of the two-phase receive. A message sent to a C+- process will contain that header, which specifies that this message should be handled by the user_message_handler, plus sections to route the message through the CPM process and finally to the destination C+- process. Program 5.3 illustrates the class-derivation tree for a C+- user message.

Within the HEAD function defined for C+- processes, an instance Outgoing_cpm_msg is newed. A buffer is allocated for the message and the constructors for the class in the derivation tree are called. Each of these constructors initializes the appropriate segment of the message. The HEAD function returns a pointer to the word in the buffer following the initialized headers. After the message variables are written into the buffer, the TAIL function for C+- processes is called, which calls the enqueue_msg function provided in the MADRE interface. The runtime system eventually injects the message into the network

Assume a block B is freed.

1. The index In of the block is extracted from the block header.

2. B can be immediately appended to the free list corresponding to index.

OR

3. While the buddy of B is free, the block can be coalesced with its "buddy."

   (a) merge the two blocks to form block B′,

   (b) In = In+1.

   Append block B′ to the free list for index In.

Program 5.2: Buddy-System Memory Deallocation.

by computing DXDY and setting MSP and MSL to point to the outgoing message (section 2.1.5). MADRE runtime-system messages are not queued, but rather are injected into the network as soon as possible (section 5.2.3).

5.2.3 Kernel Processes

As outlined in section 4.1.4, an individual runtime system may be tailored to the target machine configuration and to the spectrum of applications that will be executed. For a 16K-node ensemble executing large, very dynamic, applications, the runtime system would likely include all available mechanisms for enhancing robustness and for distributing resource demands. For a 4-node system dedicated to executing an application that exhibits static processes and message patterns, a much simpler runtime system should be constructed. Runtime-system capabilities are encapsulated within kernel processes. By defining the set of kernel processes, we construct the desired runtime system.

In this section we describe each of the kernel processes that have been developed as

```
class Outgoing_handler_msg
{ public:
            int             length;
            int             partition;
            NODE            sender;
            HANDLER*        dest_handler;
            FUNC            function;

            Outgoing_handler_msg();
};

class Outgoing_usr_msg : public Outgoing_handler_msg
{ public:
            MSG             *next,*prev;
};

class Outgoing_cpm_msg : public Outgoing_usr_msg
{ public:
            PROCESS         *dest;
            FUNC            function;
};
```

Program 5.3: Class Derivation of C+− Messages. Within the HEAD function defined for C+− processes, an instance Outgoing_cpm_msg is newed. A buffer is allocated for the message and the constructors for the class in the derivation tree are called. Each of these constructors initializes the appropriate segment of the message.

part of this thesis project. If all of these processes are included in the runtime system, the result is the most reliable runtime system we know how to construct. In the discussion of each kernel process, we describe alternatives that vary the robustness and efficiency of the composite runtime system.

Code Handler

The *code handler* provides access to the code that comprises the user's application program. Recall that a user process begins executing in response to a message. Clearly, the code that the process executes must be resident on the same node as the incoming message and the process itself. The confluence of these *three* entities – code, process, and message – is the catalyst for execution.

In previous operating systems for multicomputers, the requirement that the code be resident with the process has been satisfied in one of two ways. First, the code for a process is made resident on a node when the process is created, *eg*, Reactive Kernel [33]. If a copy of the code does not already exist on the destination node, the code is copied from the host. This approach is not practical for fine-grain machines, since hundreds of nodes could simultaneously request code from the host, creating a significant communication bottleneck.

The Cantor runtime system avoided the code-replication problem by assuming that the entire user program resides on each node. Using this assumption, a new process could be placed anywhere in the ensemble without any dynamic loading of code. This approach may be acceptable for embedded applications running on small ensembles, but is impractical for general fine-grain multicomputers. Any "real" application would likely include more code than could physically reside in the node memory. Even if the program could be loaded, the thousands-fold duplication of code would waste a large fraction of the total memory of the ensemble. Runtime systems for fine-grain multicomputers must use more sophisticated mechanisms to limit the amount of code duplication and dynamic code copying.

The first step of the fine-grain code-management strategies in this thesis is to *split the user code into pieces.* The C+– programming system includes a software tool for splitting

```
processdef P
{ public:
        atomic void f();
        atomic void g();
};

processdef Q
{ public:
        atomic void x();
};
```

Program 5.4: Declaration of Sample User-Program Code Pieces.

user programs so that the code for each atomic function is a code piece (section 3.1.4). The existing code-splitting tool includes code for functions that are not atomic in the code for atomic functions that call them. Future versions may generate individual code pieces for each non-atomic functions. (For a full description of this code splitting, see [34].) Partitioning the code for a process type by atomic function decreases the granularity of elements that the runtime system must manage, leading to more flexibility in runtime-system algorithms.

Since the entire C+- program is linked together, each code piece can be assigned a unique index. The code handler uses some mechanism – in the MADRE prototype, a simple table – to associate the code-piece number with the location of that piece. Program 5.4 and Figure 5.7 illustrate the correspondence between a user program and the MADRE code-table data structure. Arriving messages specify the index of the code piece to be executed. The dispatching process uses this index into the code table to locate the required atomic-function code.

The code table is the mechanism that prevents the duplication of code pieces on a given node. Since each access to a code piece is via the table, at most one copy of a code

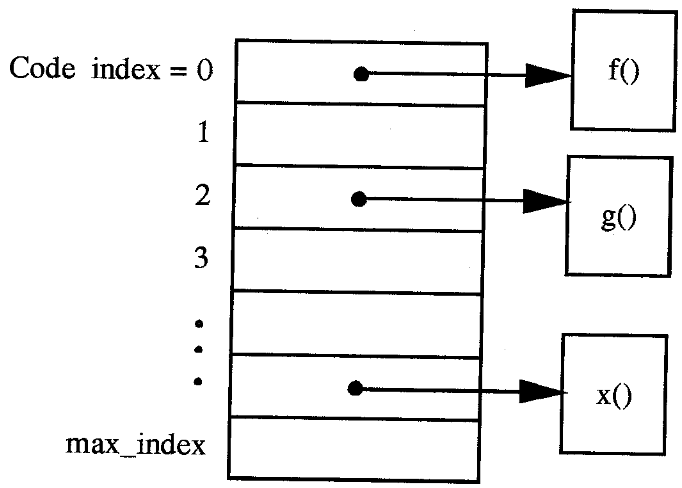


Figure 5.7: Code Table. The code handler on each node manages a table, indexed by the code-piece number, that contains the location of each piece. Arriving messages specify the index of the code piece to be executed. The dispatching process uses this index into the code table to locate the required atomic-function code.

piece will reside on the node. For those code pieces physically resident on the node, the code-table entry can contain the memory address of the code. For code pieces that are remote, an algorithm must be specified for achieving the confluence of the code piece, the message, and the process.

LRU Code Management In the Least-Recently-Used (LRU) Code Management algorithm, the code handler maintains two parallel arrays. The **code_table** array contains memory-address pointers for resident code pieces, and pointers to a special **lookup** function for remote code pieces. The **remote_code_table** array contains a node identifier for each remote code piece indicating where the nearest copy of the code can be found. The code handler is defined to include a loading procedure to fill all the code-table entries, and a procedure for accessing remote code pieces.

The host loads a user program by sending code pieces to the code handler on a subset of the nodes of the ensemble according to some mapping (section 5.3). When a code handler receives a code piece, it fills the code-table entry with the memory address of the code. In addition, it sends a **propagate_code_address** message to the code handlers on each of its four neighbor nodes. This message includes the index of the code piece, the sending node's identifier, and the direction and the cumulative Euclidean distance to the sending node. When a propagate message is received, the code handler may update the appropriate **remote_code_table** entry to reference the sending node. Each **propagate_code_address** message propagates through the mesh of nodes until it is consumed by a code handler that already has reference to a closer node that has a copy of the code piece, or it reaches the edge of the mesh.

The general approach for LRU code management is that remote code pieces are retrieved when needed, *ie*, copied from a remote node, and remain on the requesting node as a *temporary* code piece until the space they occupy is required for other memory uses. The initial placement of resident code pieces persists throughout the entire computation.

When a message arrives for an atomic function, the **code_table** is used as a jump ta-

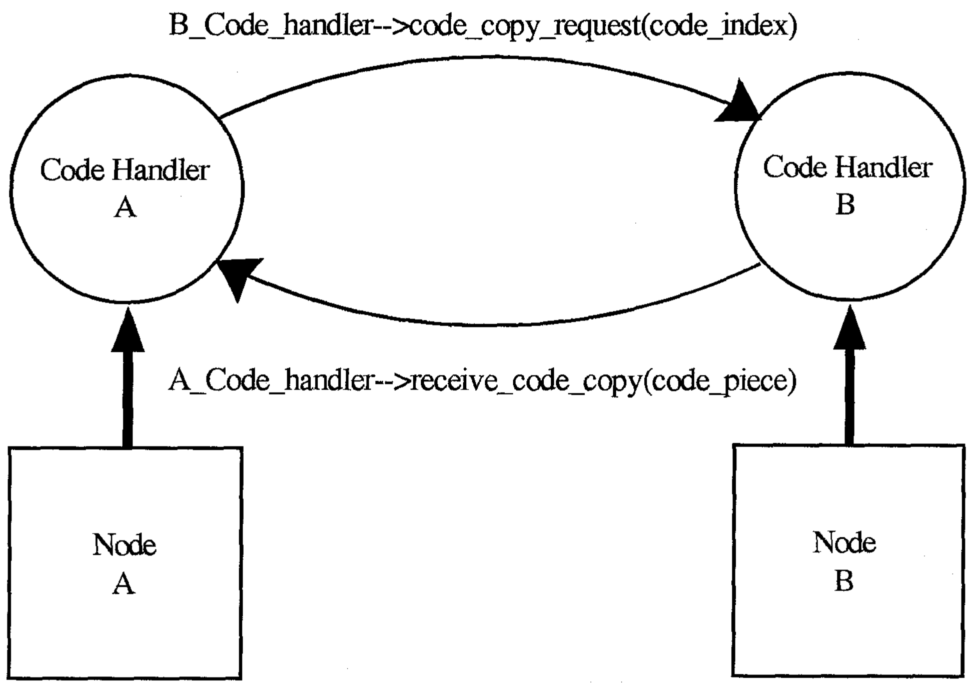


Figure 5.8: Code Retrieval for LRU Code Management. A **code_copy_request** is sent to the code handler on a remote node B that maintains a resident copy of the desired code piece. The B code handler responds with the requested code piece using the **receive_code_copy** atomic function. After the **receive_code_copy** atomic function has been executed, the required code piece is now available so the suspended user process resumes executing.

ble to begin executing the code that consumes the message. If the atomic function code is not resident, the **lookup** function will execute. Note that the execution of this function is indistinguishable from the execution of a user atomic function. Conceptually, the destination user process is executing the destination atomic function, but the code for that function has been replaced by this special function. For LRU code management, the **lookup** function uses the **remote_code_table** to locate the nearest node that maintains a permanent copy. It then sends a **code_copy_request** to the code handler on that node (Figure 5.8) and *suspends the user process using the RPC.* The code handler on the remote node sends the requested code piece to the code handler on the requesting node using the **receive_code_copy** atomic function. The remote code handler then sends an RPC reply to the suspended user process. Since message order is preserved, when the user process resumes executing, the **receive_code_copy** atomic function has been executed so the required code piece is now available. The user process begins to execute the user function code.

A linked list records the order of access to temporary code pieces – the least-recently-used temporary code piece is the rear of this list; the most-recently-used is the head. When the memory of a node becomes congested, temporary code pieces are deleted, beginning with the least-recently-used.

The LRU code-management strategy is essentially conventional code caching. The concept of having some nodes maintain permanent copies code pieces while other nodes request temporary copies is reminiscent of the *librarian* entities in [19]. This approach has been used on medium-grain machines, and will be evaluated for fine-grain machines in Chapter 6.

Remote Code Execution The confluence of message, process, and code is required for process execution. The LRU code-management strategy brings the code to the message and the process. The Remote Code Execution strategy *sends the message and the process to the code.* This strategy is motivated by the fine-grain programming model and architecture. For fine-grain programs, the amount of process data is often significantly less than the amount of code for each atomic function of that process. In those cases, copying the process data and the message is less expensive than copying the code piece. The low message latency of the fine-grain architecture permits experimentation with code-management strategies that are *not* based on code caching. For medium-grain machines, the cost of communication dictates that the code be retrieved infrequently and kept resident on the node as long as possible. With inexpensive communication, we can afford to access the code remotely more frequently.

The initialization for this code-management strategy is the same as for LRU code management. The host loads the user-program code pieces into a subset of the ensemble nodes. Each node maintains a **code_table** array that contains the memory addresses of resident code pieces, and pointers to the special **lookup** function for remote code pieces. The locations of resident code pieces are propagated through the mesh. Each node maintains a **remote_code_table** that contains the nearest node that maintains a copy of a remote code piece.

When a message arrives for an atomic function, the destination user process jumps through the **code_table** and begins executing. If the code is resident, the user process is executing the correct user-code piece; if the code is remote, the user process is executing the **lookup** function. For this code-management strategy, the **lookup** function locates the nearest node that contains a resident copy of the code piece using the **remote_code_table** array. It then sends an **execute** message to the code handler on that node (Figure 5.9). This message contains the code-piece index, the message to be consumed, *and* a copy of the state of the destination user process. The **lookup** function then suspends the user process using an RPC.

The code handler on the remote node, upon receiving the **execute** message, schedules the enclosed "shadow" user process to execute by enqueuing the accompanying "shadow" message. After the function executes, the code handler sends an RPC-reply message, containing the return values of the function and the state of the "shadow" process, back to the suspended user process. When that process resumes executing (still in the **lookup** function), it updates its state to that of the "shadow" process, and then returns any values returned

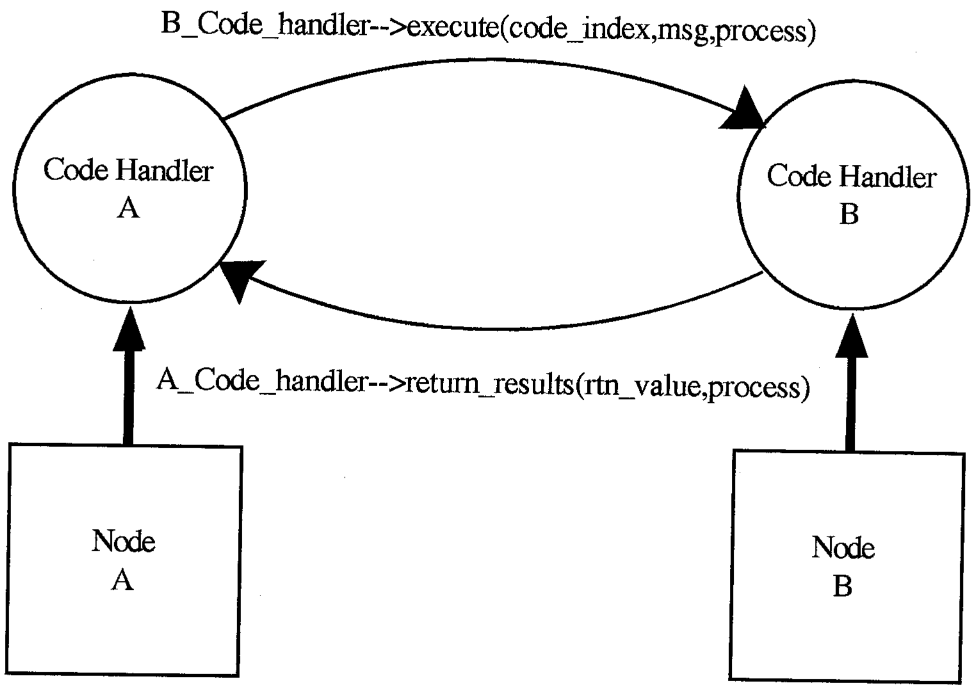


Figure 5.9: Remote Code Execution Strategy. An execute message, which contains the message to be consumed and a "shadow" copy of the destination user process, is sent to the code handler on a remote node B that maintains a resident copy of the desired code piece. The B code handler causes that code piece to be executed *on node B*. The B code handler returns whatever results the code piece returned, and it returns the state of the process, which may have been modified. The state of the suspended user process is updated to the returned process state.

by the function to the original invoking process.

The Remote Code Management strategy is certainly more ambitious than LRU Code Management or other more conventional strategies. If it performs well, we have illustrated the algorithmic flexibility of the fine-grain architecture. The low cost of communication has a tremendous impact on the set of runtime-system algorithms. In addition, the Remote Code Management strategy is an excellent vehicle for experimenting with "blurring the boundaries" between nodes. In this strategy, one node is actually allocating, for its own use, some of the processing resources of a remote node.

EXPORTED-MESSAGE HANDLER

The *exported-message handler* process is a primary part of the robustness mechanisms of the MADRE system. This handler, in conjunction with the Remote-Process Handler (page 111), frees space in the node by exporting *user* messages. A message is exported by encapsulating it within a remote process and then instantiating that process on a remote node. The exported-message handler can later invoke an atomic function of the remote process to retrieve the message for processing. Just as remote code execution is an experimental method for allocating remote *processing* resources, exporting messages is an experiment in allocating remote *memory* resources.

MADRE Robustness Mechanisms The importance of consuming incoming messages was described in section 4.1.2. The simple analogy is that the memory available on a node is a bucket with a certain capacity. Messages flow into the node like water into the bucket. As the bucket fills to capacity, some water must be removed from the bucket if more water is to be added. Otherwise, the bucket overflows. For medium-grain machines, the memory "bucket" was conceptually infinite. For fine-grain machines, the memory "bucket" is too small for that questionable abstraction. Active measures must be taken by the MADRE system to free memory so that incoming messages can continue to be received. With finite resources available on the ensemble, even these measures will eventually fail. The goal is to execute the computation until a large fraction of the total ensemble resources are exhausted, or the computation terminates.

There are two basic types of messages in the MADRE system: *system* and *user* messages. For communication deadlock to be avoided, a node must not be permanently blocked from

- allocating space for receiving an incoming message of either type,
- deallocating space by consuming the message,
- allocating space for sending an outgoing message of either type,
- deallocating space by transferring the message into the communication network.

The robustness mechanisms in the MADRE system are based on the following claim: *MADRE system messages are not permanently blocked from entering the message network.* If this claim holds, then the system messages that export user messages from the node will eventually leave the node. These export messages free space so that incoming user messages can be written into memory. The space occupied by user messages is deallocated when the message is consumed by the destination process, or the message is exported to another node.

To ensure that MADRE messages can eventually be written into memory, an empty buffer must eventually be available. The SYS partition is sized so it includes at least one buffer large enough to hold any incoming system message[2]. Since the MADRE system consumes system messages immediately rather than queueing them, this incoming-message buffer will eventually be free. The SYS partition also includes buffers for writing outgoing messages.

Thus, incoming system and user messages can eventually be written into memory. Outgoing system and user messages then cannot be permanently blocked from leaving a sending node. If the robustness claim holds, user messages can eventually be sent and

[2]Remote-process new messages are written directly into the PROC memory partition. If space is not available in that partition, the new message must be received and forwarded in fragments.

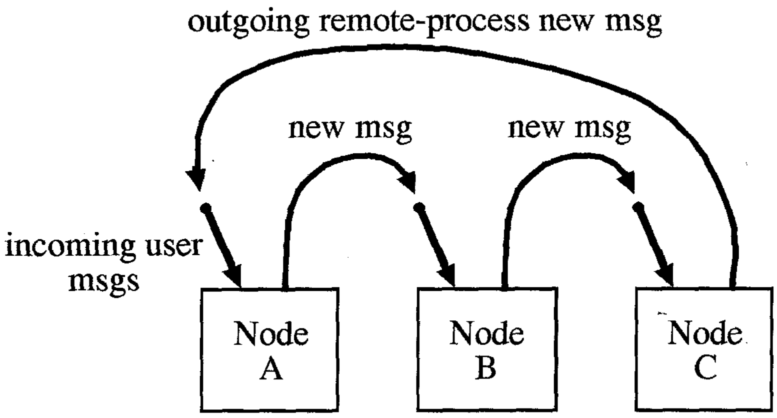


Figure 5.10: Deadlock of Message Exportation. Each node has a congested receive queue. To receive the incoming user message, each node must export a message by newing an exported-message remote process. If the destination node is blocked from receiving the user message because it too cannot export a message, the consumption property is violated and a communication deadlock has occurred on this subset of nodes.

received, and system messages can be eventually sent and received, so the consumption assumption is satisfied.

The key to satisfying the robustness claim is to ensure that outgoing system messages will eventually enter the message network. These outgoing messages could be permanently blocked only if some other message is currently being sent, and the messages filling the network channel are permanently blocked from being received.

Consider Figure 5.10. Each node has a congested receive queue. To receive the incoming user message, each node must export a message by **new**ing an exported-message remote process. This **new** message will be queued at the destination node *behind* an incoming user message. In the worst case, the **new** message itself cannot entirely leave the sending node until the destination node receives its incoming user message. If the destination node is blocked from receiving the user message because it too cannot export a message, the consumption property is violated and a communication deadlock has occurred on this subset of nodes.

A user message is the only incoming message whose reception may depend on the sending of another message, specifically a remote-process **new** message. The deadlock situation occurs because each node in the cycle must send a remote-process **new** message in order to receive a message. On page 111, we argue that the likelihood of this deadlock can be reduced by ensuring that, if it occurs, the cycle includes a large fraction of the nodes in the ensemble. On page 116, we present a handler process that implements a message-passing protocol that can be used to reduce arbitrarily the risk of this form of message deadlock.

Exporting a Message

The RCVQ Partition The purpose of exporting a message is to free space to guarantee that an incoming message can be received. For the current MADRE system, neither user processes nor runtime-system tables can be exported. Exporting an outgoing message is illogical (and unnecessary since the send queue will eventually empty if the destination nodes can receive). Thus, a user message is the only computational element that can be exported if the node's memory is congested.

The creation of the RCVQ partition was motivated by the need for message exportation. If a partition contains only user messages, then, conceptually, the entire contents of the partition can be exported to free space. If incoming user messages, processes, and tables were interspersed through the node memory, it would be possible to export all the messages on the node and still not have freed a large enough buffer for the incoming user message.

The minimum size of the RCVQ partition is fundamental to the correct operation of the message exportation algorithm. Assume the largest message that will be received is n words. Recall that a user process may be executing in the USER context when the message-exportation algorithm begins executing in the SYSTEM context. Exporting and then retrieving the message that the user process is currently consuming would be useless. If the current user process is consuming the reply from an RPC, that message is also required for continued processing of the user message. Consequently, the RCVQ partition must contain **three** buffers of size greater than or equal to n — one buffer for the active user message, one for the active RPC-reply message, and one for the incoming message. If the RCVQ partition meets this size requirement, the claim is that as long as MADRE can send a message, the consumption property is satisfied. An incoming message will always be received.

If the compiler can extract and provide information about the size of the largest message in the computation, the RCVQ partition could be sized exactly. If the largest message is much larger than an average-size message, this algorithm can be quite wasteful. Again, the runtime system can be more efficient if the user computation is expressed using small messages. In general, the performance of the computation can be improved if the RCVQ partition is larger than the minimum. Exporting and retrieving messages is certainly more costly than leaving messages *in situ* on a node. However, devoting memory resources to enlarging the receive queue means that less memory is available for user processes, thus limiting the size of the computation that can be performed.

The Exportation Mechanism Incoming user messages are queued in doubly-linked list (Figure 5.11). If the node memory is not congested, all of these messages may reside on the node in memory buffers. The pointer to the next or prev buffer is then just a memory pointer. If, however, the node memory becomes congested such that no free buffer exists to receive an incoming user message, space in the RCVQ partition must be freed via the exported-message handler.

The buffer to be cleared is determined using the memory-partitioning tree. For example, assume that node memory had been partitioned using Figure 5.5 and a buffer of size 3 is required, but currently is unavailable. The exported-message handler traverses the tree until it finds a block of size greater than or equal to 3. If that block (or any of

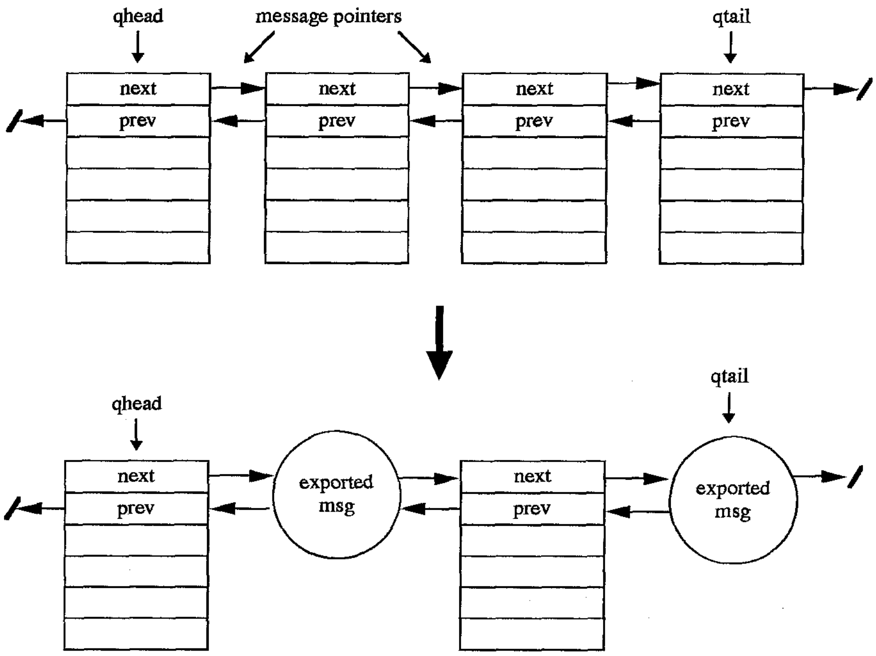


Figure 5.11: The Doubly-Linked Queue of User Messages. If the node memory is not congested, all of these messages may reside on the node in memory buffers. If, however, the node memory becomes congested such that no free buffer exists to receive an incoming message, some messages in the queue may be exported.

```
int
up_from_left(BLOCK *block, int block_size,int size)
{ if (block is large enough and does not contain the RPC_reply_msg)
      walk down the tree starting at block
   else
      continue walking up tree from block
}

int
up_from_right(BLOCK *block, int block_size,int size)
{ if (block is large enough and does not contain the RPC_reply_msg)
      walk down the tree starting at block
   else
      walk up tree from block
}

int
walk_up_tree(BLOCK *block, int block_size, int size)
{ if (block is a left block)
      walk tree up_from_left (parent_ptr,block_size, size)
   else
      walk tree up_from_right (parent_ptr,block_size,size)
   return(block_found);
}

int
walk_down_tree(BLOCK *block,int size)
{ if (block is large enough)
      if block is occupied
         export message to clear block
   else
      export occupied messages starting at block, until enough space is cleared
}
```

Program 5.5: Traversal of Memory Allocation Tree to Export Messages.

```
int
make_rcvq_space(int size)
{ if (a user message is active)
  { walk up the tree starting at the last exported msg (if not NIL) or walk down from the root
    return (block_cleared_flag)
  }
  else
  { walk up the tree starting at the active user message
    return (block_cleared_flag)
  }
}
```

Program 5.6: Traversal of Memory Allocation Tree to Export Messages. (cont.)

its sub-blocks) contains the two "non-exportable" messages (the active user message or an active RPC-reply message) the block cannot be cleared and the tree search continues. *Since there are defined to be at least three blocks of the size of the largest message, this search will eventually succeed.* Programs 5.5 and 5.6 outline the technique used to traverse the memory partition tree.

Program 5.7 and Figure 5.12 illustrate the definition and operation of an exported-message remote process. When a message is exported, the message pointers within the queue are updated. For example, next→prev and prev→next are updated to contain the reference value of the new remote process. If the next or prev are remote processes, the message pointers within those remote messages are updated by invoking a link atomic function in the remote process.

When a message must be retrieved, the retrieve atomic function of the remote process that holds the message is invoked. The remote process returns the message by invoking the returning_msg atomic function of the exported-message handler. The exported-message handler relinks the returned message into the message queue by updating the message pointers of the next and prev messages.

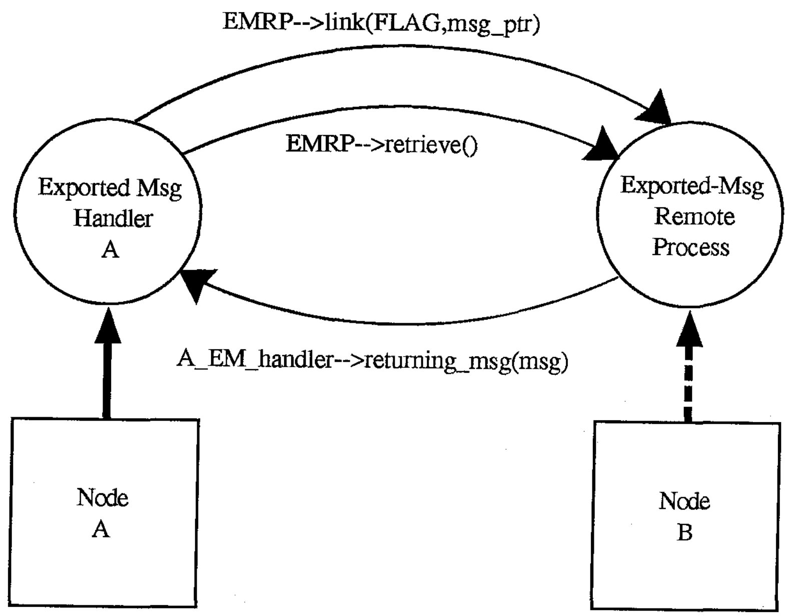


Figure 5.12: Exported-Message Process Operation. When a message must be retrieved, the **retrieve** atomic function of the remote process that holds the message is invoked. The remote process returns the message by invoking the **returning_msg** atomic function of the exported-message handler. The exported-message handler relinks the returned message into the message queue by updating the message pointers of the **next** and **prev** messages. Node B does not directly dispatch to the exported-message remote process but must indirectly invoke the process through the remote-process handler.

```
processdef Exported_msg : public Remote_process
{ Exported_msg *next;
  Exported_msg *prev;
  MSG msg;
  public:
        atomic void link(int flag,Exported_msg* msg);
        atomic MSG retrieve();
};
```

Program 5.7: Process Definition for Exported-Message Remote Process.

The message exportation scheme used in the MADRE system is fundamentally based on the queue-problem solution detailed in section 1.2. In that solution, new queue elements were encapsulated within processes that were then appended to a list of processes. For message exportation, messages in the interior of the queue can be transformed into processes. Only the operations to update the message pointers are complicated by this generalization. The crucial PUT mechanism of the queue result ensures that link messages are received before a retrieve atomic-function message.

Remote-Process Handler

The exported-message handler relies on the two services provided by the *remote-process* handler: instantiation of remote processes, and delivery of messages to remote processes. For example, in Figure 5.12, Node B does not directly dispatch to the exported-message remote process but must indirectly invoke the process through the remote-process handler. Program 5.8 outlines the declaration of the remote-process handler.

Remote Process Creation As mentioned in section 1.2.2, the *reactive* creation of remote processes is critical. Imagine the following situation. An incoming user message cannot be received, and a buffer that can be cleared is identified. A remote-process new operation is performed so that the message can be exported. A reference value of this new remote process will be required before the message can be retrieved. If the new operation for a remote process returns a reference value to the parent, how will *that* message be received when there is already a blocked incoming message at the node? The essence of the message-exportation mechanism is that messages are flushed out of the node immediately. No references can be returned to the creating process as part of the mechanism.

Since there is no returning reference value to the new remote process, there must be an indirection mechanism that can be used to refer to the process. In the MADRE system, the parent process of a new remote process assigns a unique identifier to the new process *before* placing the process. How these unique identifiers are generated is arbitrary. In the MADRE system, each node maintains a counter of the remote processes that have been created. A unique identifier (modulo counter wrap-around) is formed by combining the creating node number and the current value of this counter. This identifier is then treated as an ordinary reference value, *eg*, it can be passed between processes in messages. Atomic functions of the remote process are invoked through this identifier. By decoding this identifier, remote-process handlers can locate the remote process (see next section), thus providing an *indirect* reference mechanism.

In addition to assigning an identifier, the parent process determines the node on which the new process should be placed. The parent process sends the **new** message to the remote-process handler on that node. If the node cannot accept the remote process (*eg*, due to space constraints), the **new** message is bounced to another node (Figure 5.13). If that node cannot accept the message, it selects another node for placement and then forwards the **new** message to it. In the MADRE system, nodes are selected for remote-process placement *deterministically.* Specifically, each **new** message emanating from a particular node will first invoke the **instantiate_remote_process** atomic function of the remote-process handler on the *same* remote node (called **next_neighbor**). If the placement algorithm can be locally computed, no information must be stored about how to forward the message. This approach then ensures that a **new** message will bounce from node to node until it locates a node that can accept it. Since no node memory is consumed by this search (each stop at a node leaves no residue), the placement of a remote process can continue indefinitely without consuming

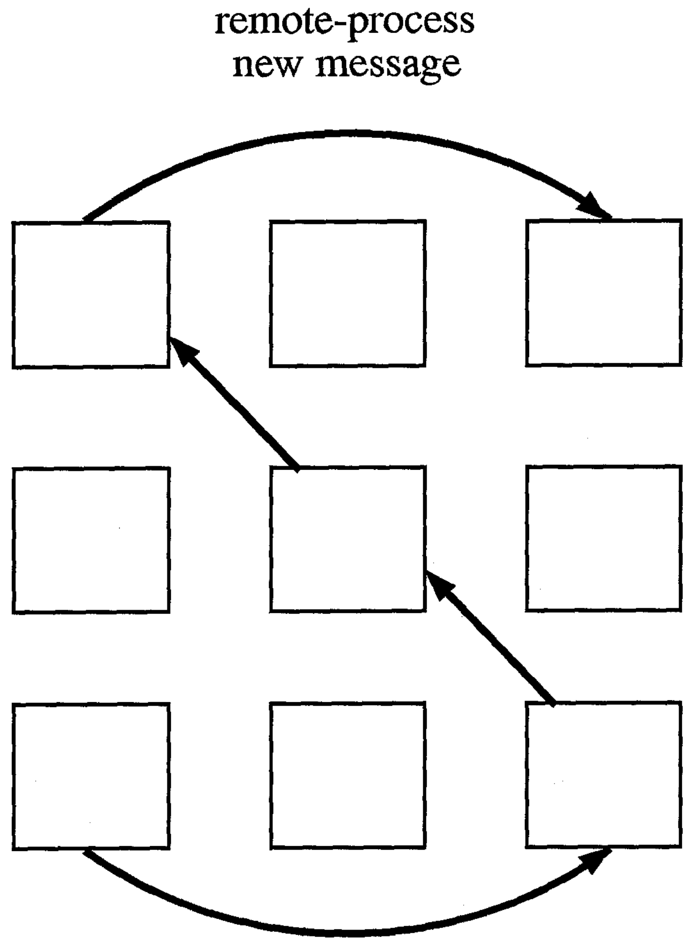


Figure 5.13: Forwarding of a Remote-Process New Message. Each **new** message emanating from a particular node will invoke the **instantiate_remote_process** atomic function of the remote-process handler on the *same* remote node called **next_neighbor.** A **new** message will bounce from node to node until it locates a node that can accept it. Since no node memory is consumed by this search (each stop at a node leaves no residue), the placement of a remote process can continue indefinitely without consuming precious memory resources.

memory resources.

In the prototype MADRE system, **next_neighbor** is provided by the host program at load time. The algorithm currently used is simply a particular offset, or *stride* away from the parent node. For example, in Figure 5.13, the stride between nodes is two. The stride should be chosen so that the vast majority of nodes of the machine are touched before a node is touched twice. If a **new** message traverses a complete cycle and returns to the original node, that remote process should be placed on the disk connected to the ensemble. By making the cycle of nodes as long as possible, we reduce the likelihood of the message-passing deadlock described on page 102.

Thus, the remote-process creation mechanism is the implementation of the reactive **new** mentioned in section 1.2.2. Since no reference value is returned from the **new** operation, no selective receive of messages is required. This mechanism is crucial to the MADRE system and to the implementation of Remote Procedure Calls. There are two primary characteristics that restrict the use of this mechanism. First, all messages sent to remote processes traverse the entire chain of nodes that were touched during placement. For exported-message remote processes, at most three messages (**new**, **link** and **retrieve**) traverse the chain of reference so this algorithm does not introduce significant overhead. However, using this algorithm for user-process placement would be inefficient since an unknown number of messages might follow a potentially long chain of references. Instead, using the implementation of an RPC provided by the remote-process-creation mechanism, user processes can be placed and a reference value returned. All subsequent communication to the process is therefore direct.

The second disadvantage is that there is no built-in indication of when the remote-process instantiation has completed. Assume that the reference to a new remote process is communicated to another process and that process sends a message to the remote process. The order of arrival of this message and the **new** message is nondeterministic. If the **new** message is not received first, an error has occurred. If the remote-process references are to be used by processes other than the creating process, some form of synchronization is

```
processdef Remote_process_handler: public Handler
{ int *hash_table;
  int sys_proc_ident_cnt; // counter used to produce identifiers

  public:
        atomic void deliver_msg(ID destination,MSG msg);
        atomic void instantiate_remote_process(ID
                            destination,ARGS args);
};
```

Program 5.8: Process Definition for the Remote-Process Handler.

required to ensure that new operation has completed. This synchronization is not, however, required for each remote-process new. Instead, many remote processes can be created, with a single completion signal indicating when the entire structure has been instantiated.

Handler Operation When a process sends a message to a remote process, the destination identifier must be decoded by the sender to determine which node is the first in the chain of reference. Since the destination identifier is an indirect reference to the process, the message is sent via the remote-process handler.

In the MADRE system, a hash table of identifiers within the remote-process handler is used to locate the remote process. Each index in the table points to a linked list of remote processes that are resident on the node. When the deliver_msg atomic function is invoked (Program 5.8), the remote-process handler hashes the destination identifier into the fixed set of indices. Using the resulting hash index, the corresponding linked list is searched for the remote process. If the process is not found in the list, the message is forwarded to the next_neighbor node, which may house the remote process.

```
processdef Reply_handler : public Handler
{
  public:
          Ack ack_sndq_start[NUM_ACKS];
          Ack *ack_sndq_limit;
          Ack *ack_sndq_head;
          Ack *ack_sndq_tail;

          atomic void handle_ack();
          atomic void handle_nack();

                    Reply_handler();

                    void send_ack() ;
                    void send_nack();
          void enqueue_ack(int type,NODE dest);
};
```

Program 5.9: Process Definition for the Reply Handler.

REPLY HANDLER

The *reply handler* is an optional handler that illustrates how handlers can be included to experiment with different software implementations of message-passing protocols. The reply handler included in the maximally-robust MADRE configuration implements a two-trip, or *receipt-acknowledged*, message-passing protocol. This algorithm is based on the sending node maintaining a copy of the message until the message has been safely received by the destination node. This protocol is used for user messages in the maximally-robust configuration of the MADRE system.

This message protocol has stronger message-order-preservation properties than conventional one-trip protocols. By waiting for an acknowledgment to a message sent from process A to process B, we are assured that B receives that message before any messages sent subsequently from A, through a third process, C, to B. If found to be useful, this protocol could be incorporated into future versions of Mosaic message-passing hardware.

Program 5.9 outlines the definition of this handler. The receiving node invokes the

handle_ack atomic function of the sending node's reply handler if a message has been received and written into **RCVQ** memory. If the receiving node cannot receive the message, it invokes the **handle_nack** atomic function and the incoming message is flushed by being received into a preallocated buffer that is used in write-only mode. In the case of a large incoming message and a smaller flush buffer, the message will be received, and then flushed, in fragments.

Note that the outgoing **handle_ack/handle_nack** acknowledge messages must exit the node as conventional messages, so they may be temporary blocked from entering the network. However, the receiving node continues to receive and/or flush messages, generating more **handle_acks** or **handle_nacks**, that must be queued for sending.

The acknowledge queue is a statically defined queue in the **SYS** memory partition. The outgoing acknowledge messages themselves are not queued, only the type of the acknowledge (**handle_ack** or **handle_nack**) and the destination node for the acknowledge. In the worst case, every node has sent one message to a particular node. In that case, the destination node may need to queue N acknowledges. Since N may be as large as 16K, this approach could require more memory than is available on the node. Fortunately, this worst case is highly improbable. *By increasing the size of the acknowledge queue, we reduce the likelihood of the message-passing deadlock described on page 102.* In practice, a small acknowledge queue of order 10 has been used reliably for Mosaic ensembles with 256 or fewer nodes.

handle_ack/handle_nack acknowledge messages are small, requiring only a header that indicates the destination reply handler and destination atomic function. This header can be completely received on the destination node during the first phase of the message receive (*ie*, into the **msg_hdr_buffer**). Thus, these *acknowledge messages can always be received.* If the **handle_ack** atomic function is invoked, the message being acknowledged can be freed and the next message (if any) can be sent. If the **handle_nack** atomic function is invoked, the reply handler simply resends the rejected message. *The sending node does not send another message until the **handle_ack** to the previous message send has been received.*

Implementation The inclusion of the reply-handler in the MADRE configuration currently requires some minor modifications to the MADRE interrupt-handling routines. In the **buffer_full_interrupt** handling routine, if the allocation of a receive buffer fails, MRP and MRL should be set so that the message is received into the flush buffer. In the **receive_interrupt** handling routine, the sending node's reply-handler atomic functions are invoked: **handle_ack** if the message was received, **handle_nack** if the message was flushed.

Two flags are modified by the reply handler to ensure that a new message is not sent until the ack from the previous message has been received.

waiting_for_ack A message has been sent to which the ack has not been received. No additional user messages can be sent. Acknowledge messages (**handle_acks** or **handle_nacks**) and runtime-system messages can be sent however since they can always be received by their destination nodes.

usr_send_pending A message has been sent, but the **send_interrupt** has not been been received, indicating that the message has not yet entered the network. No messages can be sent until this interrupt is received. When the **send_interrupt** is received, the next queued acknowledge message should be sent. If the acknowledge queue is already empty, the next queued user message should be sent if not already **waiting_for_ack**. **usr_send_pending** implies **waiting_for_ack**.

USER-PROCESS HANDLER

The *user-process handler* queues incoming user messages for consumption by the CPM (C-Plus-Minus) process (see section 5.2.4). This handler is the producer of the queue of messages that is the interface between the two contexts (Figure 5.1). All handlers execute in the SYSTEM context, while user processes operate in the USER context.

Program 5.10 outlines the definition of the user-message handler. When this process is instantiated, it initializes the **qhead** and **qtail** of the message queue to NIL. The constructor for this handler also configures the USER context by setting the APC (Alternate-Program Counter) register so that the CPM process (the consumer of the message queue) will execute

```
processdef User_process_handler : public Handler
{ MSG *qhead,*qtail;
  public:
                User_process_handler();
        atomic void queue_msg(MSG msg);
};
```

Program 5.10: Process Definition for the User-Process Handler.

upon context switch.

When the queue_msg atomic function of the user-message handler is invoked, the enclosed user message must be appended to the queue of messages. If the queue is not empty, the next field of qtail message is updated either by sending a link message if the qtail message is remote, or by simply updating the memory reference. The prev field of the new message is assigned the value of qtail and then qtail is set to refer the new message.

By adding functions to the user-message handler, capabilities to trap certain user messages, such as user-process new messages, can be added. These capabilities can optimize performance since they execute in the priority SYSTEM context.

Termination Handler

The *termination handler* is an example of an optional handler that can be included in a runtime system configuration. In the MADRE system, this handler is used to detect acquiescence both of the loading phase and, later, of the user computation. Program 5.11 outlines the definition of this handler.

The distributed algorithm executed by the termination handlers on each node is an extension of Dijkstra's algorithm [14] for detecting termination. In [38], Taylor presents the distributed algorithm and a proof of its correctness. A similar termination detection algorithm is used in the Cantor runtime system.

Each node maintains a "color" variable depending on whether it is busy (BLACK) or

```
processdef Termination_handler : public Handler
{
  public:
        int num_msgs_sent;
        int num_msgs_rcvd;
        int node_color;
        int node_holds_token;
        int token_color;
        int token_sum;
                Termination_handler();
        atomic void receive_token(int,int);
                void send_token();
                void is_rts_idle();
};
```

Program 5.11: Process Definition for the Termination Handler.

idle (WHITE). In the MADRE system, a node is WHITE only if no interrupts are pending, the user-message queue is empty, and no user message is currently being processed. The node becomes BLACK whenever a **send_interrupt** or **receive_interrupt** is received for non-termination-detection messages. Each node also keeps a count variable that is the difference between the number of messages sent from this node and the number of messages received. A *token* is comprised of a color variable and a count variable.

The termination detection algorithm uses a token that circulates through the nodes in the ensemble. A phase of the algorithm begins with the host sending out a (WHITE, 0) token to a node. When a token enters a node, if the node is BLACK, the token becomes BLACK. Later when the node becomes idle, the node's count of messages is added to the token count of messages and the token is sent to the next node. If the node is WHITE when the token arrives, the count variable is added to the token count and the token is immediately sent to the next node. If two consecutive phases of this algorithm return a WHITE token with a count of zero to the host, the ensemble has reached an acquiescent state.

This algorithm does not add significant overhead to the user computation. The token propagates slowly through the nodes as they become idle and then very quickly through the nodes when the computation has terminated. In the MADRE system, we use a linear ordering of all the nodes (including the host node) for simplicity.

5.2.4 The CPM Process

The modular design of the underlying MADRE system allows direct substitution of handlers and processes that implement support for different programming languages. The MADRE components most likely to be substituted are the code handler (page 94), and the user-message handler (page 118). The CPM process (Figure 5.1) is not a MADRE process, but rather a C+−-specific process that is layered atop the MADRE system. Hence, to support additional programming notations, this process would also be replaced.

Recall that the user-message handler queues incoming user messages for consumption by user processes within the USER context. In general, the CPM process consumes the message queue by decoding the relevant message section and then invokes an atomic function of the destination user process. If the message-exportation capability is included in the MADRE system, the operation of the CPM process is more complex. The message at the front of the message queue may have been exported. *This message must be retrieved before any other user message can be processed by the CPM process.* This constraint is dictated by our choice of queuing strategy. The **next** pointer in the queue of messages resides within the message. Thus, when a message has been exported, the information about how to access the remainder of the queue is not available to the CPM process.

RPC Handling

The C+− programming notation includes the capability for a Remote Procedure Call (RPC) (section 3.2.3). However, the MADRE system does not explicitly provide an RPC to user processes, but rather provides the mechanism for guaranteeing that messages can be received (relying on message exportation). The CPM process must provide the necessary runtime support for the execution of RPCs. Relegating this function to the CPM process adheres

```
CPM::CPM
{ while (1)
    if (qhead != NIL)
    { if (qhead is local)
       { invoke destination function of user process
          if (process did not suspend)
             free space occupied by message
          if (messages had been buffered at process) and (message was consumed)
             prepend buffered messages to message queue
          if (RPC-reply message exists)
             free space occupied by RPC-reply message
       }
       else
          if (not already waiting for returning qhead)
          retrieve qhead message
    }
}
```

Program 5.12: Operation of the CPM Process.

to the layered-system philosophy. From the perspective of the MADRE processes, user messages are "consumed" as soon as they are placed into the internal user-message queue. Independent of the runtime system, the CPM process and user processes can cooperate to modify the order of message processing (*eg*, a RPC-reply message is processed before a message that was received earlier).

The algorithm executed by the CPM process is outlined in Program 5.12. In normal operation, the **qhead** message is removed from the message queue and delivered to the destination process for consumption. If the message is completely consumed, the memory occupied by the message is freed, and control is returned to the CPM which removes the next message from the queue. Assume instead that, while consuming the message, the destination process executes an RPC. In that case, the user process uses an inherited function called **wait** to suspend its own execution and return control to the CPM function. The space occupied by the message being consumed by the now-suspended user process is *not* freed.

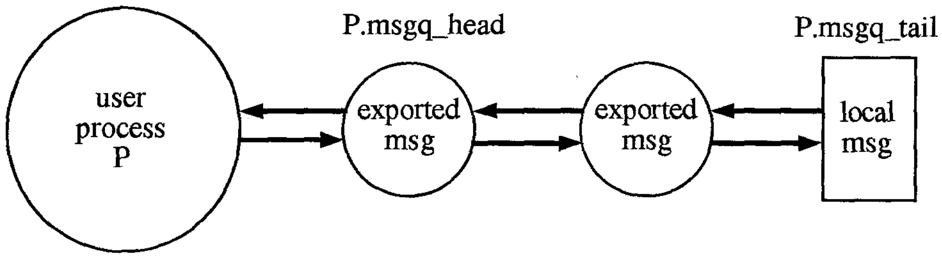


Figure 5.14: Buffered-Message Queue for a Process. If the destination process has been suspended and the message is not the reply to the RPC, or if the destination atomic function is not active, the message is appended to a local queue for that process. Note that messages buffered at the process may also be exported to free space for incoming user messages.

The CPM process decodes the destination of the next message in the queue. If the destination process has been suspended and the message is not the reply to the RPC, or if the destination atomic function is not active, the message is appended to a local queue for that process (Figure 5.14). If the message is an RPC reply, execution of the suspended destination user process is resumed. The desired returning values are extracted from the RPC-reply message.[3] When the user process suspends again, or when it finishes consuming the original message, the memory occupied by the RPC-reply message can be freed.

When a message has been consumed by a process, any messages that were buffered must be re-examined. In the case of active and passive operations, the consumption of a message may have altered whether or not a particular message will be accepted. For RPCs, when the process is revived, each of the messages that were buffered while the process was suspended must now be processed. Consequently, all messages in the process's

[3]The code for this extraction is generated automatically by the C+− compiler.

local message queue are *prepended* to the front of the message queue. The CPM process proceeds to dispatch off each of these messages as if they had never been handled. If these buffered messages have been exported, they must be retrieved. Again, our choice of queuing strategies, such that no queue information is permanently resident in the node, precludes shortcuts that allow the CPM process to identify messages that will be immediately re-buffered. Future runtime systems will likely include hybrid algorithms that keep some local information about the queue contents, without allowing the space occupied by that queue to grow arbitrarily.

5.3 Host Services and Loading

A program running on the ensemble host executes the following operations to load and execute a C+- program.

- After resetting the ensemble, the host sends a message that contains the MADRE runtime system to each node in the ensemble. Upon receiving this, the Mosaic node begins executing instructions at the beginning of the runtime-system message. The code for the runtime system includes space allocated for the statically instantiated MADRE Root and kernel processes. Ensemble parameters such as mynode and num_nodes are initialized by directly modifying the internal variables of the MADRE Root process within the message before it is sent.

- A message containing the number of user-code pieces that will be loaded is sent to the code handler on each node. This handler then allocates the code-management tables described on page 94.

- Each user-code piece is loaded onto a subset of nodes in the ensemble. In the prototype MADRE system, the host program uses simple mapping schemes for selecting nodes. Even-numbered code pieces are loaded on even-numbered nodes, or every nth code-piece is loaded on every nth node, etc. Future MADRE systems will experiment with more sophisticated code-mapping heuristics. Beginning on page 94, two of the algorithms used to allow access to non-resident code pieces are described.

- The host program initiates sends the termination-detection token to node (0, 0) of the mesh. This token circulates through each of the nodes in the ensemble to detect when all of the loading messages have been consumed (page 119).

- When the host program detects that loading has terminated, it instantiates the **root** process of the C+- program on a node in the ensemble. In the prototype MADRE system, the node selected is a node near the center of the ensemble. The host program then initiates another termination-detection cycle; this token cycle is used to detect when the user computation has terminated.

The host program includes a set of C+- processes that corresponds to the set of kernel processes instantiated on each node in the ensemble. During the execution of the C+- program, messages sent from the ensemble to the host processes are delivered to the destination handler on the host. For example, remote-process **new** messages may be sent to the remote-process handler if they could not be placed on the nodes in the ensemble (page 111).

The host program also includes an **error** process, to which ensemble processes send messages to report error conditions. The prototype MADRE system includes a single **console** process, to which ensemble processes send print messages. By increasing the number of **console** processes and initializing the **console_ref** given to each node in the ensemble, more sophisticated algorithms for concurrent I/O can be implemented.

6 Experimental Results

Our research group has developed a complete set of prototypes that span the computational levels (section 1.4):

- C+− as the high-level programming environment,
- MADRE as the fine-grain runtime system,
- and the Mosaic C as the hardware implementation of the fine-grain multicomputer architecture.

These prototypes allow the algorithms employed by the MADRE system to be evaluated experimentally. In this chapter, we present experiments in which test C+− programs are executed on a 256-node Mosaic C.

6.1 Experimental Method

The experimental method used in this thesis has been to define and then test several hypotheses concerning runtime-system design and implementation. For example, we hypothesize that a runtime system can do a good job of automatic process placement. We hypothesize that runtime-system operation can be made more robust by including message-exportation capabilities, without excessive loss of efficiency. We hypothesize that efficient code-management algorithms are required, and feasible, for fine-grain multicomputers.

We investigate these hypotheses by varying the runtime-system implementation and measuring its performance when executing a suite of test programs. Test programs have been selected for inclusion in the suite because they satisfy two criteria: their behaviors are simple enough to be immediately understood, yet they illustrate the performance differences between runtime-system algorithm alternatives.

```
processdef collector_handler : public Handler
{ public:

        int number_of_processes;
        int num_msgs_cross_bisection;
        int num_usr_msgs_sent;
        int num_msgs_sent;

        DIST_ARRAY usr_msg_length_array;
        DIST_ARRAY length_msgs_array;
        DIST_ARRAY msg_distance_array;
        DIST_ARRAY placement_distance_array;
        DIST_ARRAY hop_array;

                collector_handler();
                atomic void collect_phase(DIST_ARRAY usr_msg_length,
                                          DIST_ARRAY message_length,
                                          DIST_ARRAY message_distance,
                                          DIST_ARRAY placement_distance,
                                          DIST_ARRAY hop_distribution);
};
```

Program 6.1: Process Definition for the Collector Handler.

```
processdef Host_collector
{ public:
        atomic void rcv_node_data(int x,int y,int num_processes,
                                  int num_existing_processes,int num_usr_msgs,
                                  int num_msgs,int num_msgs_cross_bisection,
                                  int proc_memory_available,int proc_memory_size);
        atomic void rcv_vector(int usr_msg_length[ARRAY_SIZE],
                               int msg_length[ARRAY_SIZE],
                               int msg_distance[ARRAY_SIZE],
                               int placement_distance[ARRAY_SIZE],
                               int hop_distribution[ARRAY_SIZE]);

                Host_collector();
        void    print_output();
};
```

Program 6.2: Process Definition for the Host Collector.

Each program is executed for a number of Monte Carlo trials [7]. The random-number generator of the runtime system is seeded differently for each run. Executing the program repeatedly mitigates the effects of initial conditions on runtime-system performance and provides a more accurate estimate of output distributions. For each program, we can measure distributions on variables such as the number and locations of processes that are created, the number of user messages sent, the size of the messages, the memory utilization of each node, and the distance traveled by user messages. These measurements are performed by adding a *collector handler* to the set of kernel processes on each node (Program 6.1). This handler includes counter variables that are updated throughout the execution of the program.

A *host collector* process (Program 6.2) is included on the host to assimilate data from the ensemble. When the user computation terminates or halts due to an error, the host program initiates a data-collection phase by invoking the collect_phase atomic function of the collector handler on node $(0, 0)$. The collector handler on that node replies immediately to the host-collector process using the rcv_node_data atomic function. The arguments to this function include the performance data that pertain exclusively to that node, such as the number of processes as a function of (x, y) coordinates. The collector then invokes the collect_phase atomic function of its next_neighbor. This message contains the cumulative values of ensemble performance data, such as the distribution of user-message lengths. Eventually, the ensemble replies to the host collector with the cumulative data using a rcv_vector message.

6.2 Process Placement

The choice of process-placement algorithm is a crucial aspect of runtime-system design and implementation. The overall execution performance of the user program is directly dependent on the placement of processes, particularly for large diameter machines such as a 16K-node Mosaic ensemble.

6.2.1 Algorithms

The modular design of the MADRE system allows the direct substitution of different process-placement algorithms by defining two functions, pick_node and failure_pick_node. The pick_node is called by the parent process to select a node for placing a new process. The failure_pick_node function is called by the runtime system when a process-placement attempt fails. Process-placement attempts may fail due to memory congestion on the selected node. The failure_pick_node function uses a threshold function to determine when process placement should be re-attempted or the computation should be halted. In the current system, the number of placement attempts is one of the arguments in a new message. When this *hop count* exceeds a predefined limit called hop_threshold, the computation is halted.

By defining separate mechanisms for initial process placement and for process-placement failures, we can experiment with placing processes using one algorithm when the machine is lightly loaded and another algorithm when resources become more scarce. There are three algorithms used in this thesis for selecting nodes for process placement.

random A node in the ensemble is selected at random by computing a random number modulo the number of nodes in the ensemble. No locality between the parent and child processes is preserved using this algorithm. Placing processes purely at random provides a useful base case for the class of randomized process-placement algorithms. The practicality of this algorithm has been demonstrated by Athas in earlier work [4].

walk One of the four neighbors of the node where the parent process resides is selected at random. If the placement attempts repeatedly fail, this algorithm in effect executes a random walk of the machine. This algorithm can vary by whether the edges of the mesh reflect the random walk, or they allow the walk to "wrap-around" to the opposite side of the mesh. In contrast to purely random placement, this algorithm represents the other endpoint of the locality *vs.* dispersal spectrum. This algorithm is not expected to be practical for fine-grain multicomputers as its emphasis on local placement encourages congestion of processes.

k-biased A distribution on a variable k, the distance from the parent node to the selected node, is input to the runtime system at initialization. A distance d is selected according to this distribution: a node at distance d from the parent node is selected at random. This algorithm varies according to whether the mesh edges reflect or wrap-around placement attempts and according to the type of the distribution used. Currently, the distribution of k can be either uniform, normal or Poisson. The mean and variance of the normal and Poisson distributions can also be varied.

6.2.2 Experiments

To evaluate the various process-placement algorithms, we have developed a test suite of C+− programs and the performance-measuring tools discussed in section 6.1. The program test suite currently includes a n-queens program, a prime-number sieve, a perfect-number generator and Program 6.3. Each of these programs was selected because its behavior is simple enough to be readily understandable and the problem size, *eg*, the number of processes created, can be varied easily.

```
processdef P
{ public:
      atomic void f(int x);
};

void
P::f(int x)
{   ifc→console→print(x);

      P* p = new P(x+1);
      p→f(x+1);
}

root::root(ARGS a)
{   P* p = new P(0);
      p→f(0);
}
```

Program 6.3: Process-Placement Test Program.

Program 6.3 contains the C+- code for the simplest test program used to investigate the process-placement algorithms. The behavior of this program is that a process is instantiated on a node; this process then instantiates another process. The program halts in error when the hop count of the process-placement attempt exceeds the preset threshold hop_threshold.

Program 6.3 was executed for a number of trials (typically 10) using each of the nine process-placement algorithms (random/random, random/walk, random/k-biased, etc.). Intuitively, one can predict the behavior of some of these algorithms. For example, random/random should do a good job of dispersing the process load across the machine, but will not preserve any locality between parent and child processes. Walk/random and walk/k-biased should preserve as much locality as possible when the machine is lightly loaded, and then switch to wider dispersal when nodes become congested. The k-biased algorithms should exhibit locality and dispersal that is tunable, due to the fact that the input distributions can be varied.

Effect of Increasing hop_threshold

Figures 6.1 and 6.2 contain histograms of the number of processes per node using a random-node initial placement and a random-node failure placement strategy (*ie*, random/random). Each histogram corresponds to a different value of **hop_threshold** – 0, 1, 2, 5, 10, 20, 50. For Program 6.3, the capacity of each node is about 170 processes. As the number of hops that a **new** message may traverse increases, more nodes have been filled to capacity before the program fails.

Figures 6.3 and 6.4 contain three-dimensional histograms of the occupancy of each node in the ensemble after one trial execution of Program 6.3, using random/random placement, for each value of **hop_threshold**. Intuitively, the random/random algorithm should do the best job of balancing the process load across the ensemble. Even for this algorithm, increasing **hop_threshold** results in a noticeable smoothing of the distribution of processes across the machine. Figures 6.5 and 6.6 illustrate the effects of increasing **hop_threshold** for the walk/walk placement algorithm. This algorithm is most susceptible to failure due to congestion in the neighborhood of the parent. The node-occupancy figures also illustrate that this algorithm may do a poor job of utilizing the entire machine, particularly if the edges of the mesh reflect the walk and no information about the history of the walk has been maintained.

Figure 6.7 plots the total number of processes as a function of **hop_threshold** for three algorithms that use a random initial placement. Random/random is plotted as a solid line, random/walk as a dashed line and random/k-biased-normal as a dotted line. Figures 6.8 and 6.9 contain similar plots for the walk and k-biased placement algorithms. From these curves, one can see that permitting a small number of hops in process-placement significantly improves the utilization of the machine.

Parent-Child Distance

The distance from the node on which the parent process resides, the parent node, to the node on which the child process is placed, the child node, is a good metric for evaluating how

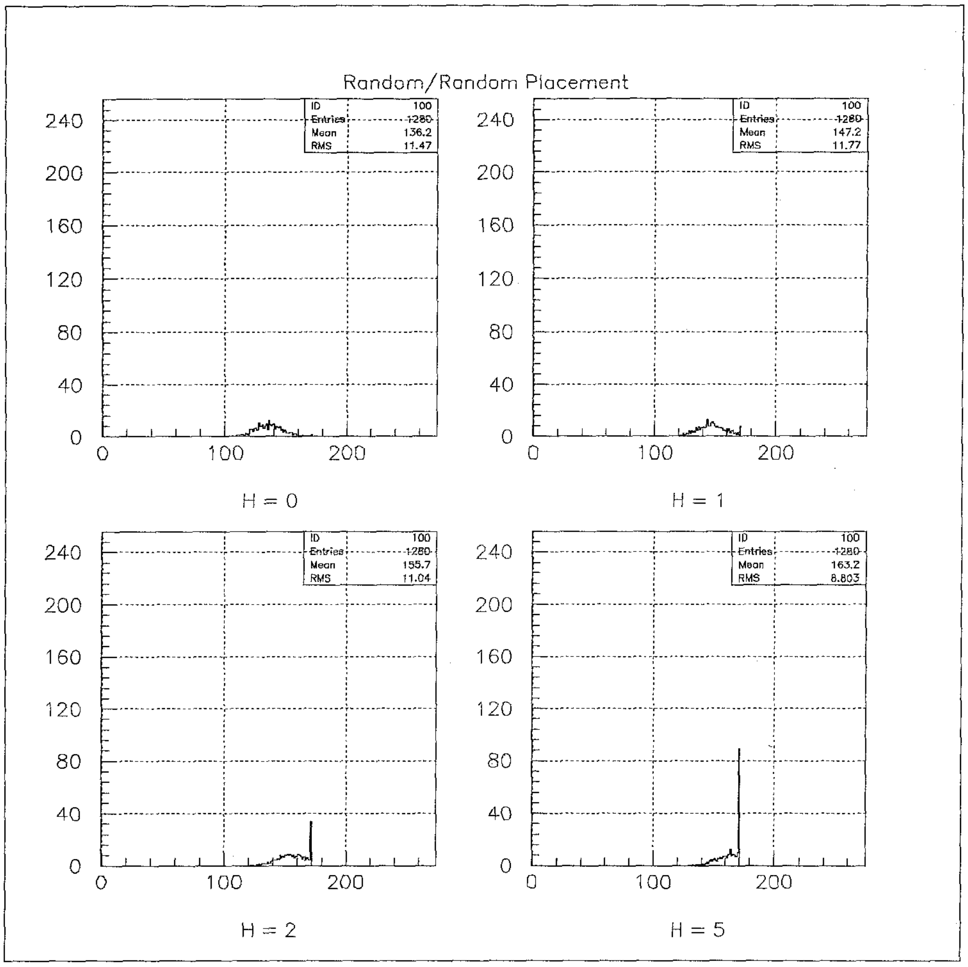


Figure 6.1: Number of Processes Per Node – Random/Random Placement. The x-axis represents the number of processes resident on a node; the y-axis is the number of nodes that have x resident processes when the computation fails.

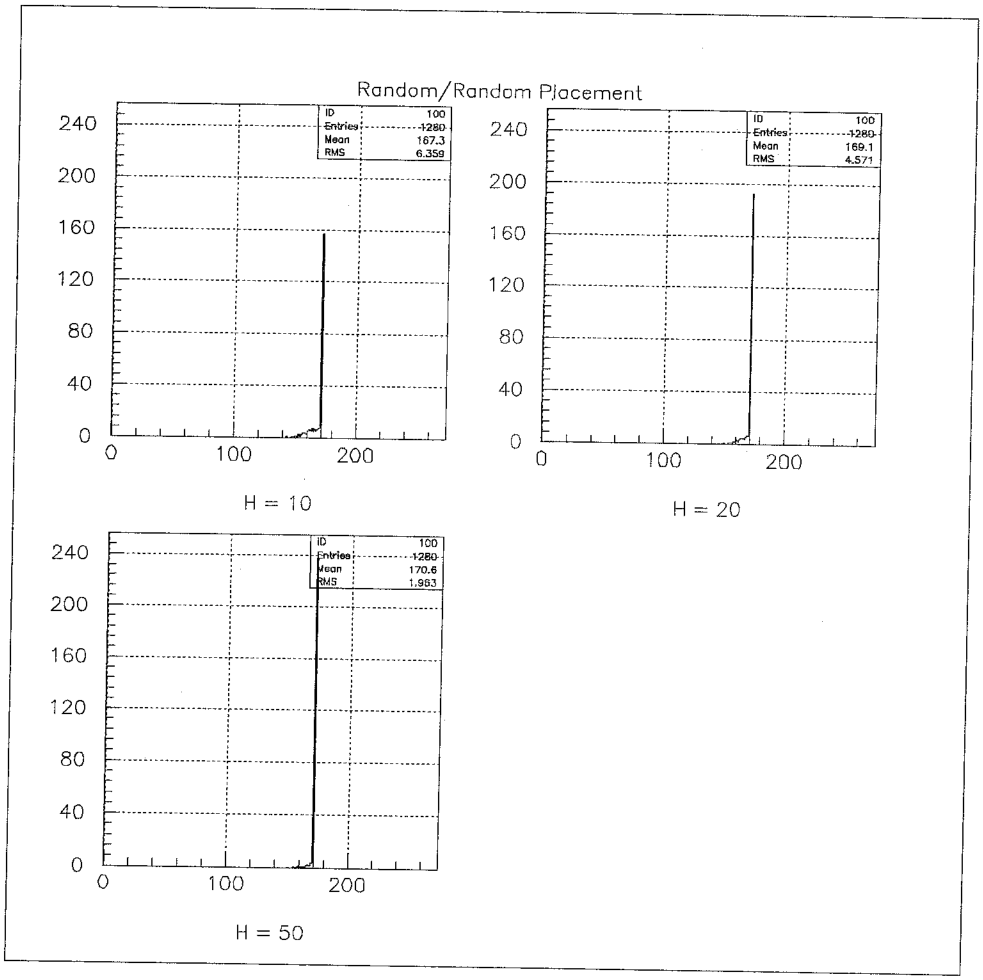


Figure 6.2: Number of Processes Per Node – Random/Random Placement (cont.).

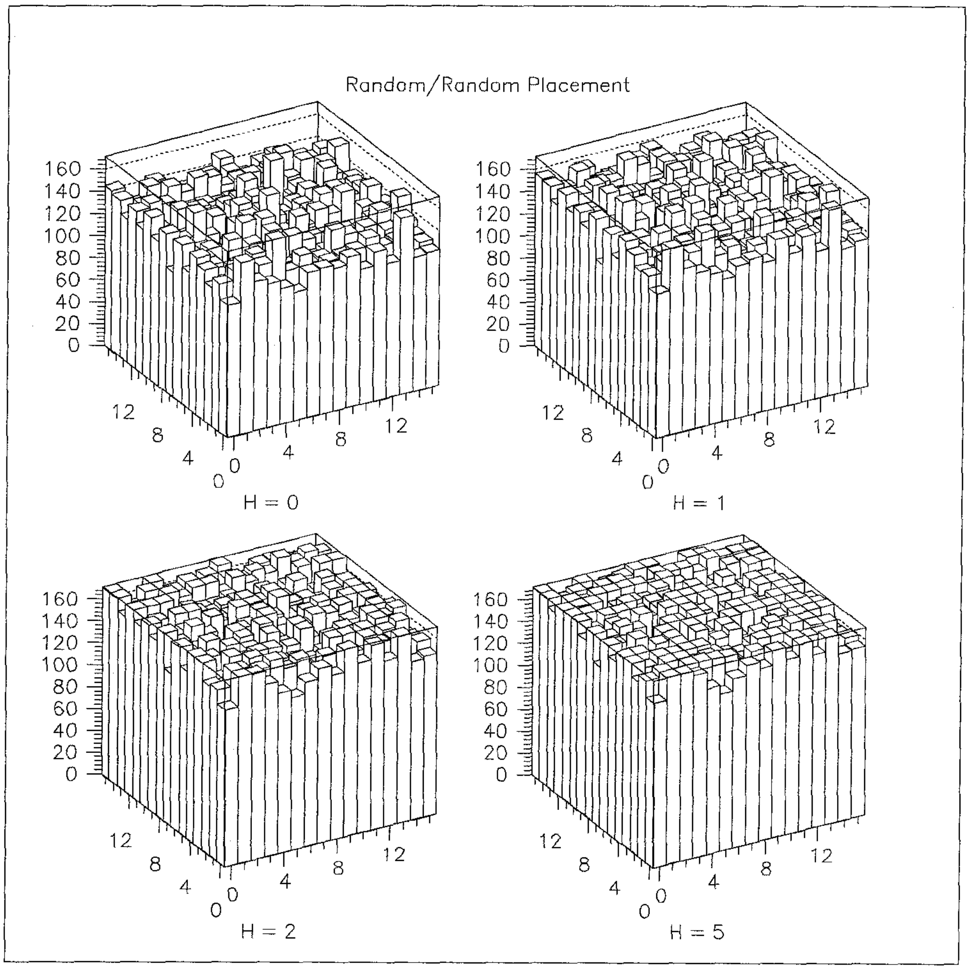


Figure 6.3: Node Occupancy – Random/Random Placement. The number of processes resident on each node is plotted as a function of the node coordinates (x, y).

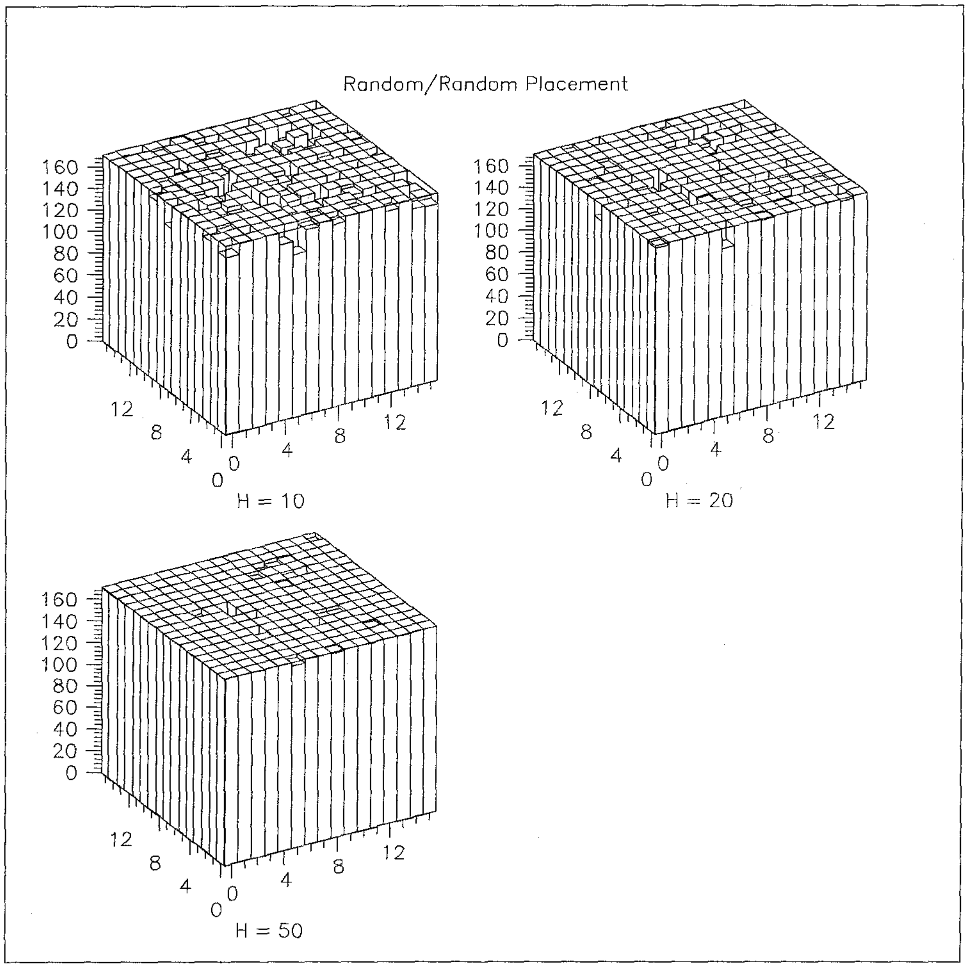


Figure 6.4: Node Occupancy – Random/Random Placement (cont.).

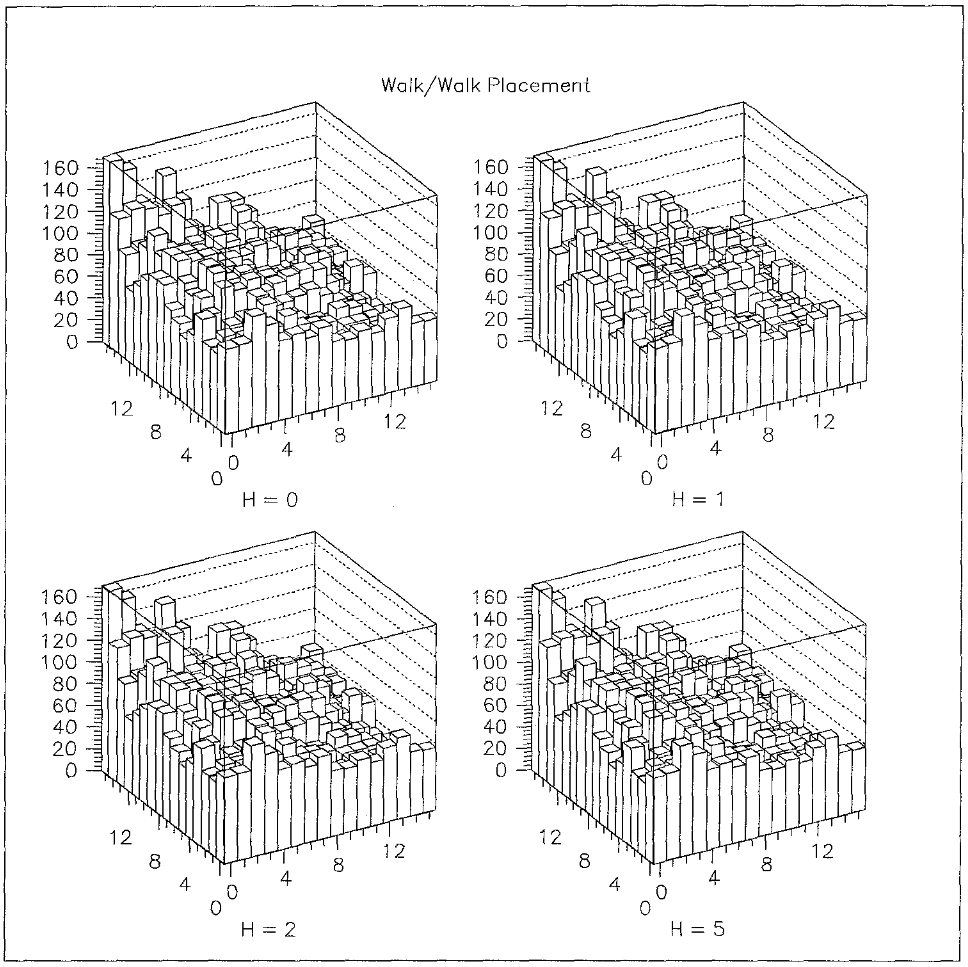


Figure 6.5: Node Occupancy – Walk/Walk Placement. The number of processes resident on each node is plotted as a function of the node coordinates (x, y).

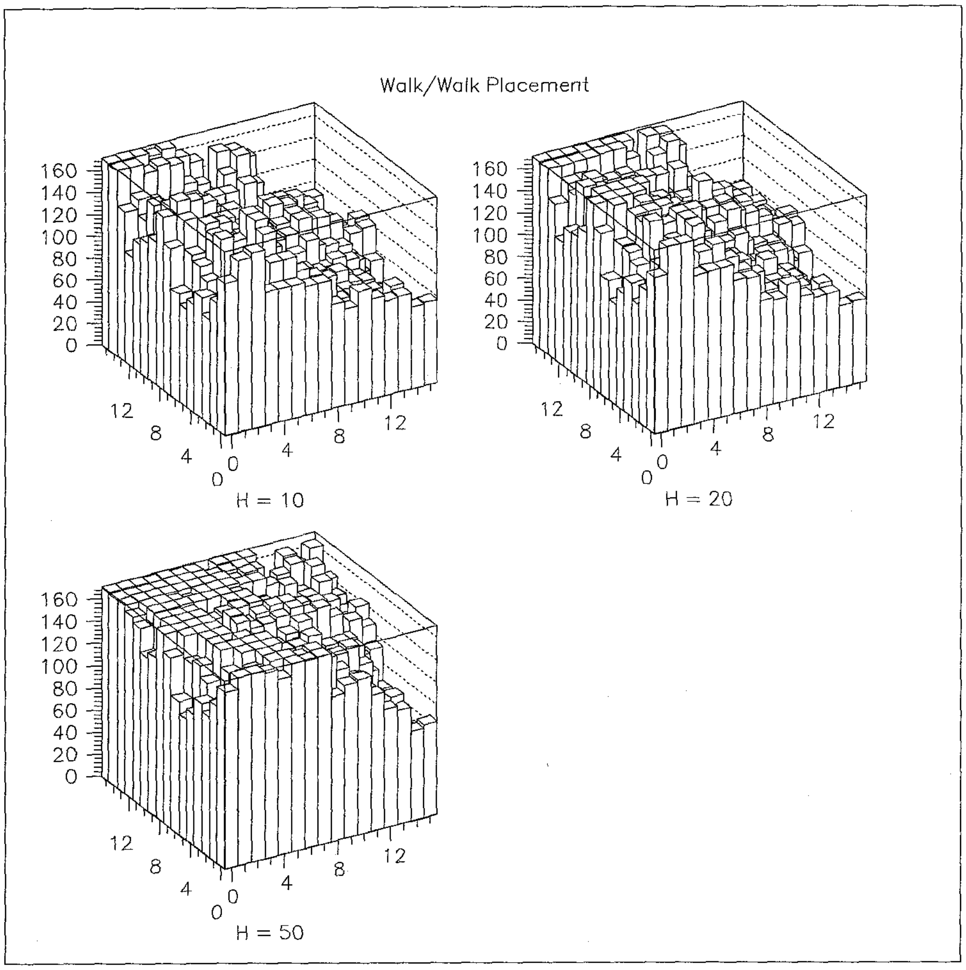


Figure 6.6: Node Occupancy – Walk/Walk Placement (cont.).

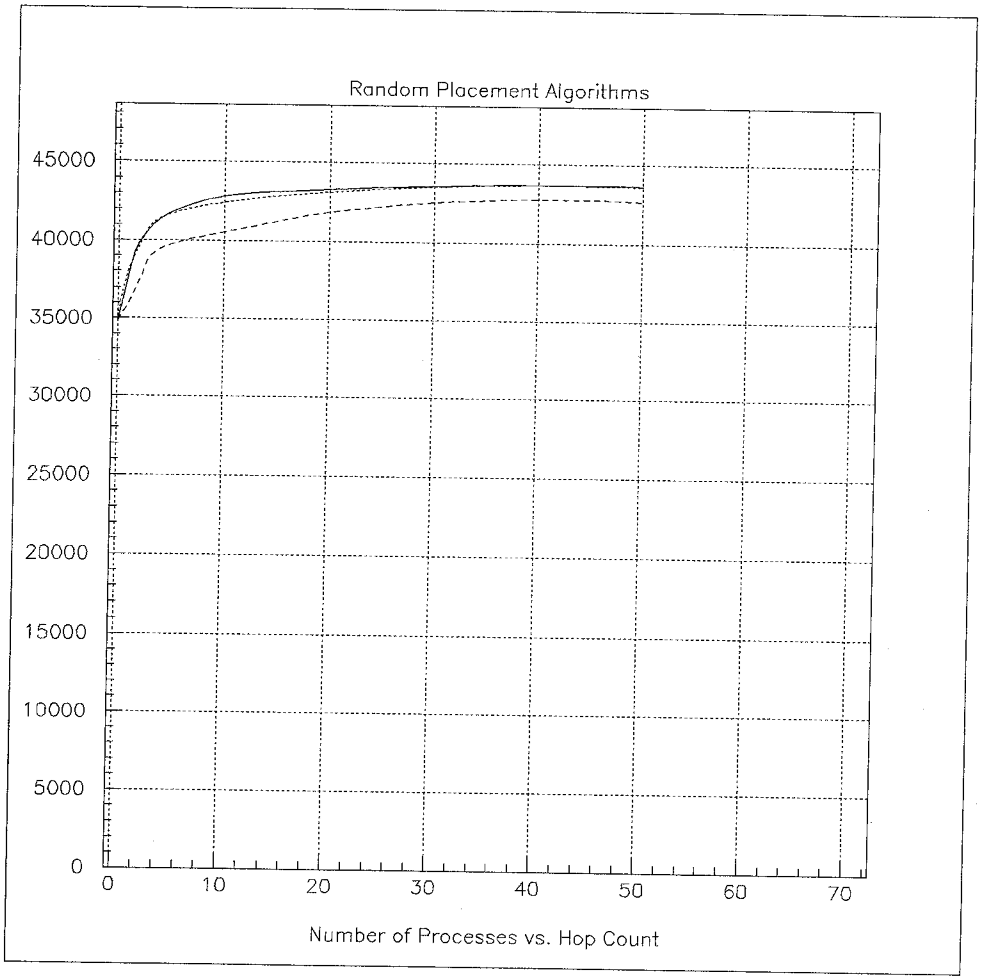


Figure 6.7: Number of Processes as a Function of Hop Count – Random Placement Algorithms. Random/random is plotted as a solid line, random/walk as a dashed line and random/k-biased-normal as a dotted line.

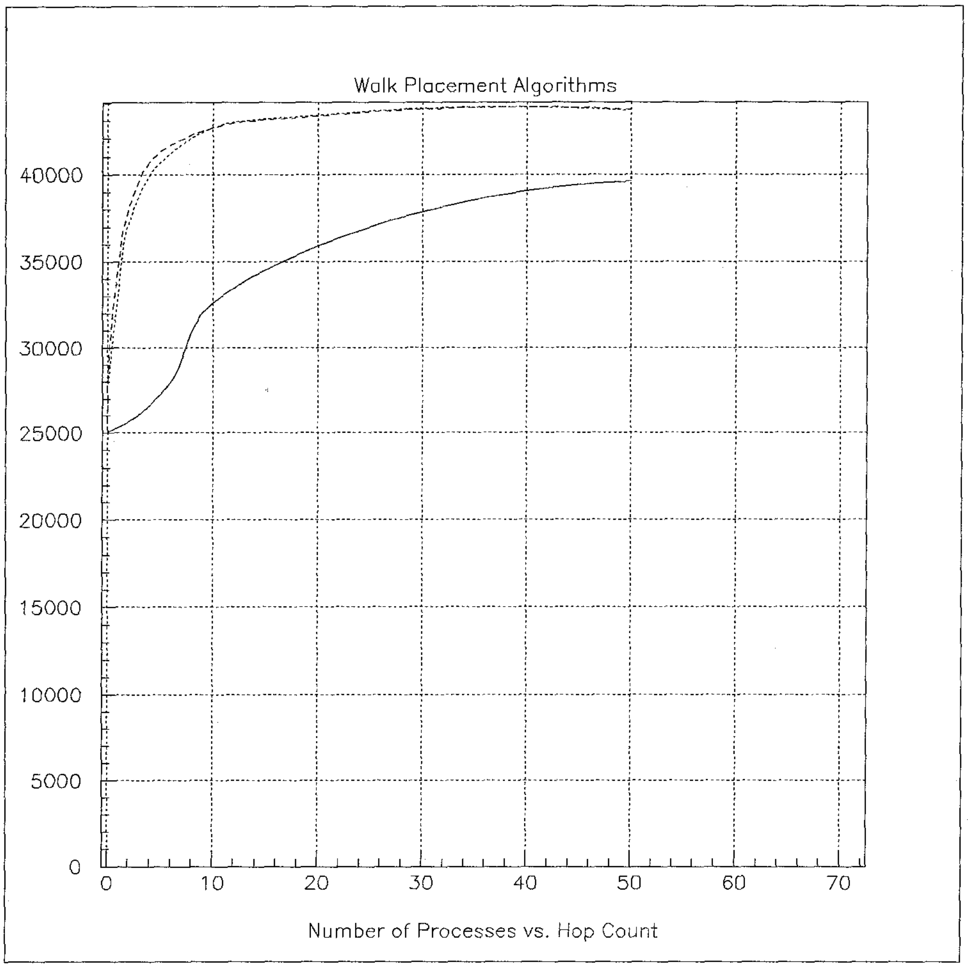


Figure 6.8: Number of Processes as a Function of Hop Count – Walk Placement Algorithms. Walk/walk is plotted as a solid line, walk/random as a dashed line and walk/k-biased-normal as a dotted line.

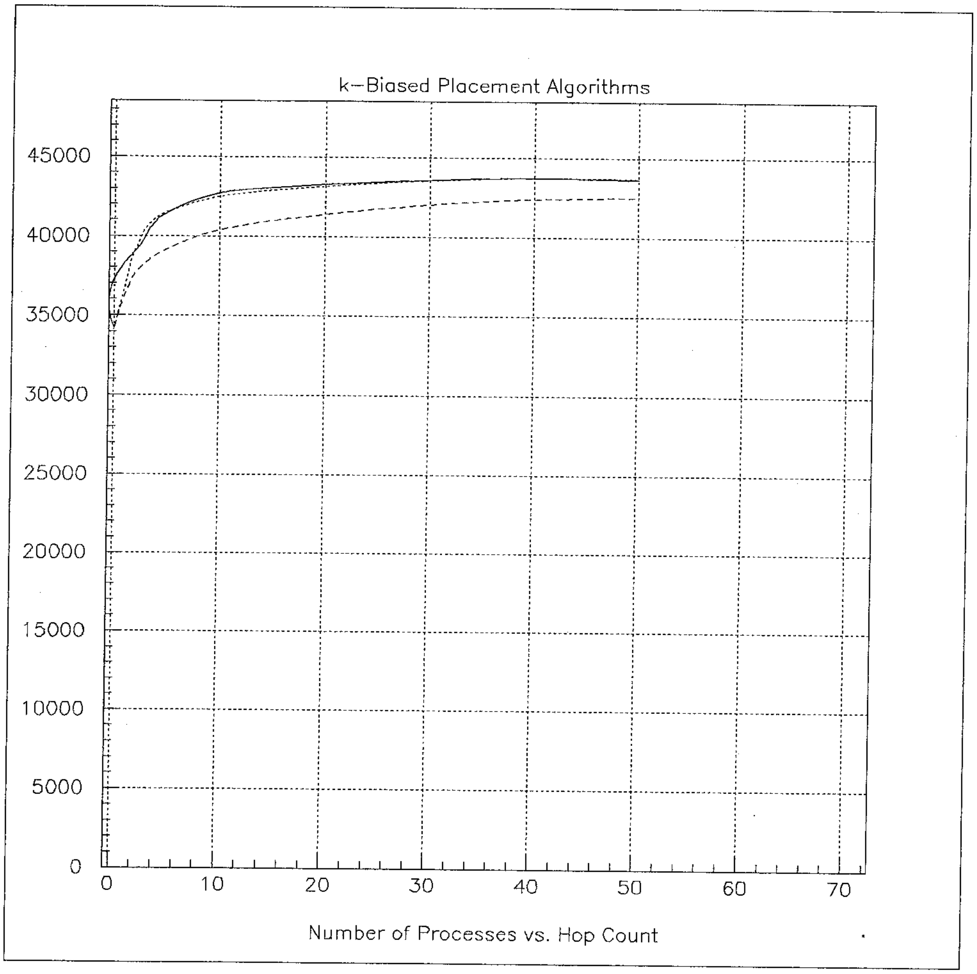


Figure 6.9: Number of Processes as a Function of Hop Count – k-Biased Placement Algorithms. k-biased-normal/random is plotted as a solid line, k-biased-normal/walk as a dashed line and k-biased-normal/k-biased-normal as a dotted line.

well a particular algorithm disperses the process load. The walk/walk algorithm minimizes parent-child distance, but at the cost of aggregating the process load near the parent node. Conversely, the random/random algorithm disperses the process load without regard for the distance from parent to child node. Figures 6.10 and 6.11 contain histograms of the distance from parent to child node using the random/random and walk/walk algorithms, respectively. The initial placement algorithm determines the shape of these curves; varying the algorithm used when process placement fails does not appreciably alter the curves.

Note that the average distance between two nodes chosen at random of an $n \times n$ mesh is

$$\frac{2}{3}(n - \frac{1}{n})$$

or 10.625 for a 16×16 mesh. Figure 6.10 illustrates that the random/random algorithm results in a mean parent-child distance of 10.64, conforming to the predicted value.

Figures 6.12 – 6.17 illustrate the parent-child distance for the uniform, Poisson, and normal distributions used in the k-biased placement algorithm. Each of these figures depicts first the input distribution used to select the distance from parent to child, and the measured distribution of parent-child distances.

The difference between the input distribution in Figure 6.12 and output distribution in Figure 6.13 is the result of choosing the input distance to be uniform over the maximum distance from one corner of the mesh to another, in this case, a distance of 32. Only the corner nodes can however place child processes at a distance of 32 without having the placement reflect off of (or wrap-around) the edges of the mesh. The output distribution in Figure 6.13 illustrates the effects of the mesh boundaries on the measured parent-child distances.

Figures 6.14 – 6.17 illustrate that k-biased family of algorithms permits the distance from parent to child node to be a tunable parameter that is input to the runtime system. The output distributions are shifted slightly toward zero in comparison to the input distributions. This effect is attributable to placement attempts that reflect off the edges of the mesh. For larger meshes, this shift should be less significant. If the placement attempts are allowed to

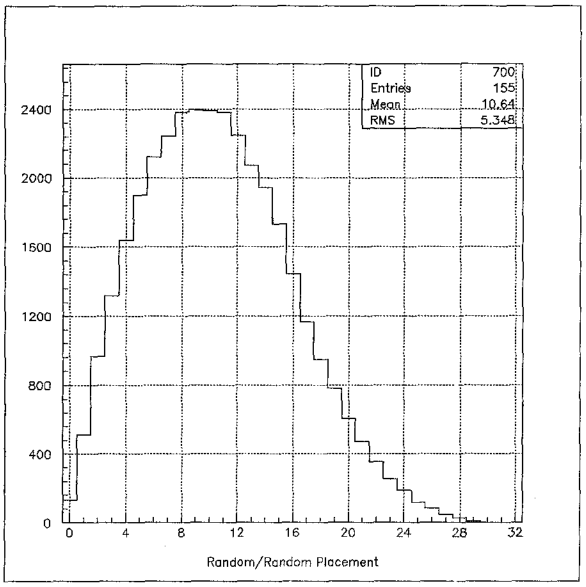


Figure 6.10: Distance Distribution from Parent to Child – Random/Random Placement.

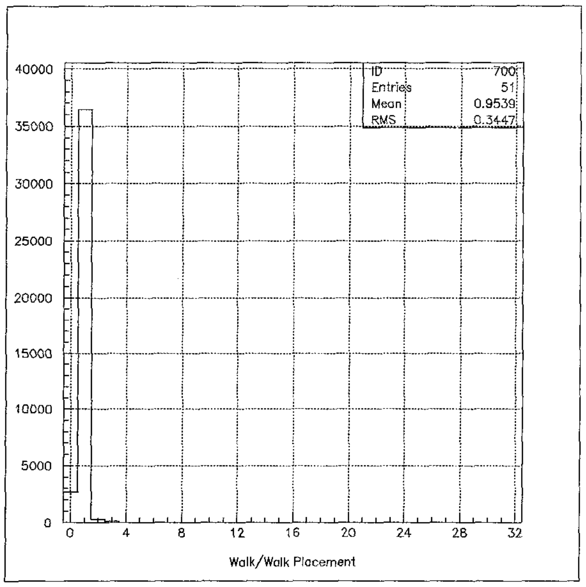


Figure 6.11: Distance Distribution from Parent to Child – Walk/Walk Placement.

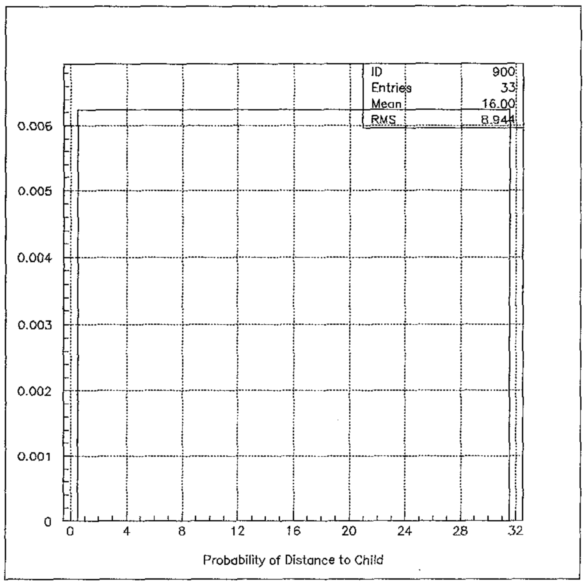


Figure 6.12: Probability of Distance from Parent to Child – k-Biased-Uniform/Random Placement.

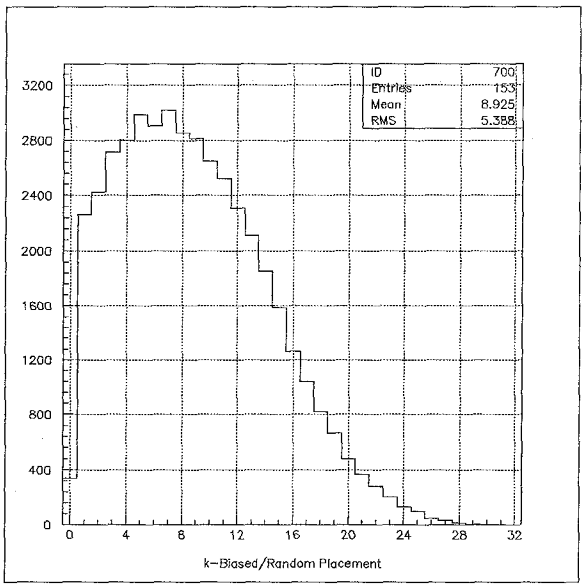


Figure 6.13: Distance Distribution from Parent to Child – k-Biased-Uniform/Random Placement.

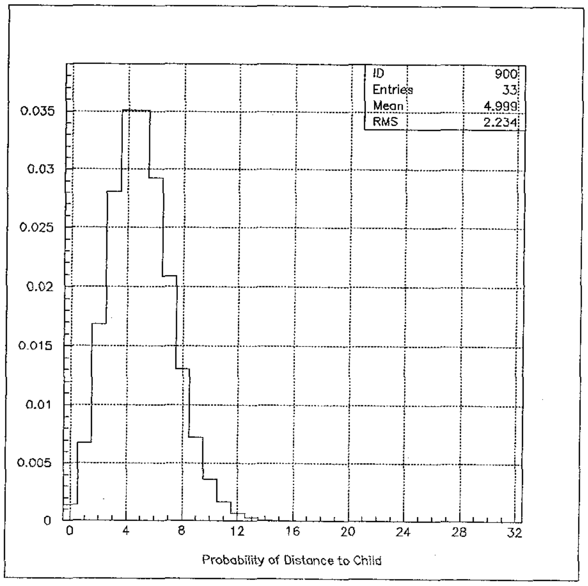


Figure 6.14: Probability of Distance from Parent to Child – k-Biased-Poisson/Random Placement.

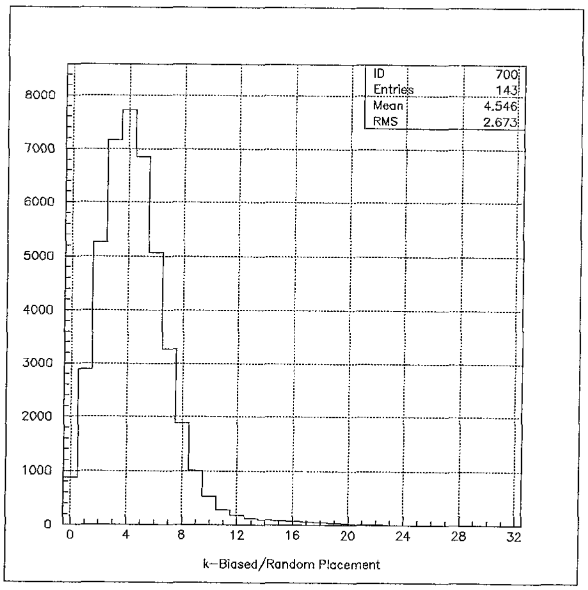


Figure 6.15: Distance Distribution from Parent to Child – k-Biased-Poisson/Random Placement.

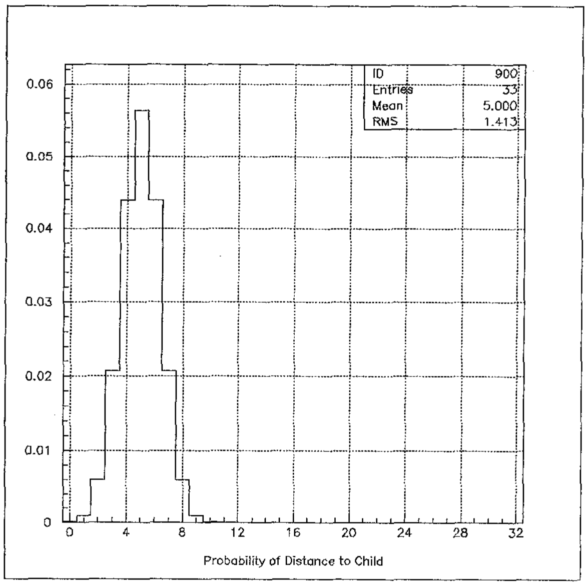


Figure 6.16: Probability of Distance from Parent to Child – k-Biased-Normal/Random Placement.

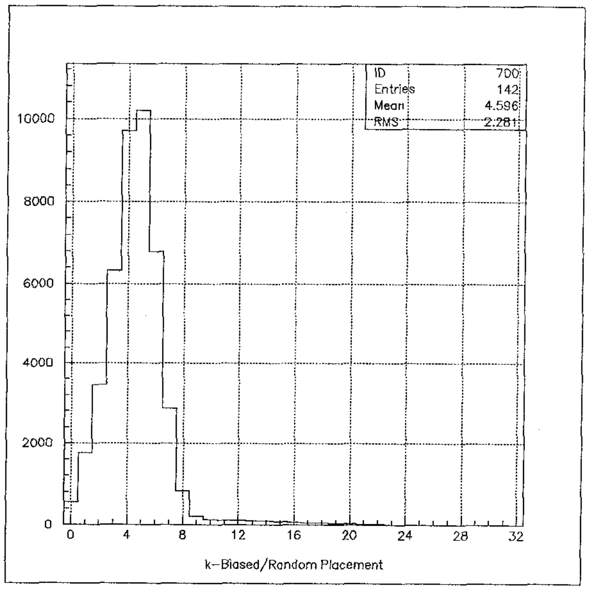


Figure 6.17: Distance Distribution from Parent to Child – k-Biased-Normal/Random Placement.

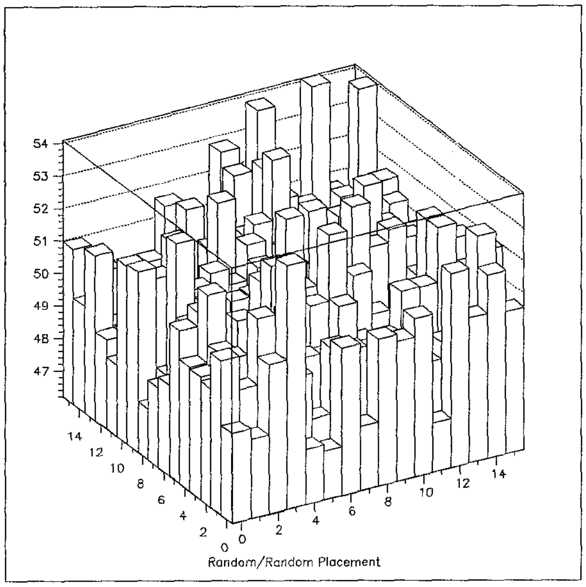


Figure 6.18: Percentage of Messages Crossing Bisection – Random/Random Placement.

wrap around the edges of the mesh rather than reflect, the output distribution is shifted to the right.

MESSAGES CROSSING THE NETWORK BISECTION

The choice of process-placement algorithm directly affects the the number of messages that cross the message-network bisection. For the experiments in this section, the Mosaic ensemble was divided in half in the x-dimension. Each node in the ensemble maintained a count of the user messages that cross this boundary.

Figures 6.18 and 6.19 contain histograms of the percentage of messages that cross the network bisection for each node in the ensemble. The first graph depicts the percentage of messages crossing the network bisection using the random/random placement algorithm. As expected, roughly half of the messages sent between random pairs of nodes in the mesh cross the network bisection. The second graph illustrates the same metric using the walk/walk algorithm. Figure 6.20 contains similar histograms of the percentage for the k-biased/random

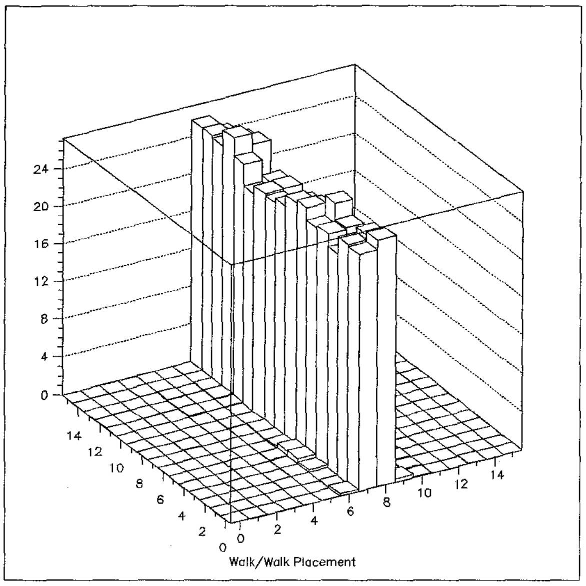


Figure 6.19: Percentage of Messages Crossing Bisection – Walk/Walk Placement.

algorithm using uniform, Poisson, and normal input distributions, respectively.

6.3 Robustness Evaluation

Although each of the test-suite programs is quite simple, ensuring the robustness of their execution has proved to be non-trivial. The test-suite programs each generate combinatorially-large trees of processes and then compute some function using that tree. As the tree is being constructed, process-placement attempts may flood into a given node in the ensemble, quickly overwhelming its receive queue. For several of the process-placement algorithms, such as the walk/walk strategy, that do not widely disperse the process-placement attempts, this problem is particularly acute. The message-exportation mechanism described in section 5.2.3 is crucial for ensuring that the computation can proceed through the initial period of building the tree of processes and execute to completion.

Analytical methods can be used to examine the probability of receive-queue overflow using the random/random placement strategy. If L is the number of process-creation re-

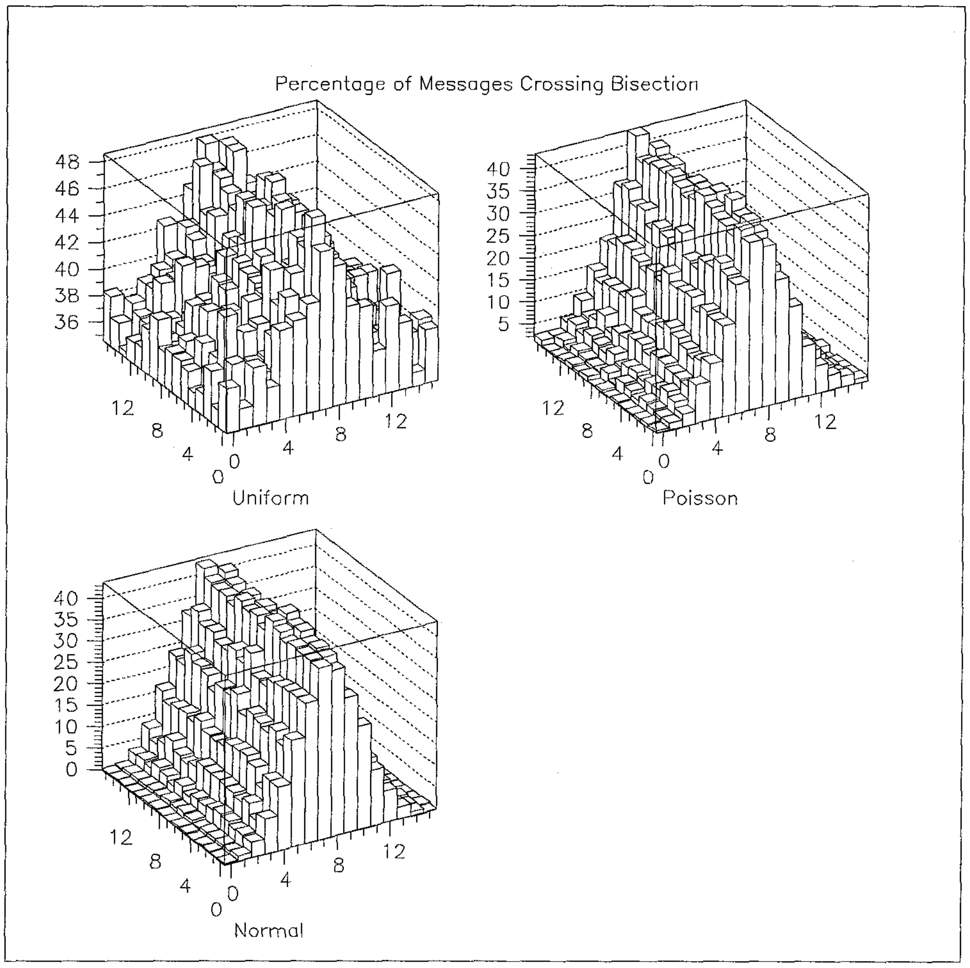


Figure 6.20: Percentage of Messages Crossing Bisection – k-Biased/Random Placement. The three histograms depict the percentage of messages crossing the network bisection as a function of (x, y) using the uniform, Poisson and normal input distributions to the k-biased/random placement algorithm.

quests that are concurrently being attempted, N is the number of nodes in the ensemble, and C is the number of such requests that can reside in the receive-queue of a node, the probability of receive-queue overflow is given by

$$P_{overflow} = 1 - \sum_{i=0}^{c} \begin{pmatrix} L \\ i \end{pmatrix} (\frac{1}{N})^{i} (1 - \frac{1}{N})^{L-i}$$

Figure 6.21 illustrates this probability for L equals 1, 2, 5, 10, and 20 times N. For each of these curves, the probability of receive-queue overflow is about 45% if the expected number of process-creation requests received by each node is equal to the receive-queue capacity. These curves demonstrate that the likelihood of receive-queue overflow is unacceptably high unless the receive queue capacity is significantly larger than the expected number of process-creation requests received. *The message-exportation mechanisms discussed in Chapter 5 effectively increase the receive-queue capacity, thus decreasing the possibility of receive-queue overflow.* For placement strategies that emphasize locality, the expected number of processes per node can be much larger than if random/random placement were used; thus making the message exportation capability crucial.

6.4 Future Experimental Work

6.4.1 Process Placement

Section 6.2 presented the experimental results of different process-placement algorithms being used to execute the simple test program, Program 6.3. These results illustrate how a tunable degree of locality can be employed in the process-placement strategy. The process-placement algorithms need to be further evaluated on the current suite of test programs plus more sophisticated application programs. Additional placement algorithms that rely on limited knowledge of the history of previous placement attempts will be included in these experiments.

6.4.2 Robustness Evaluation

Experiments to further evaluate the robustness measures of the MADRE system include:

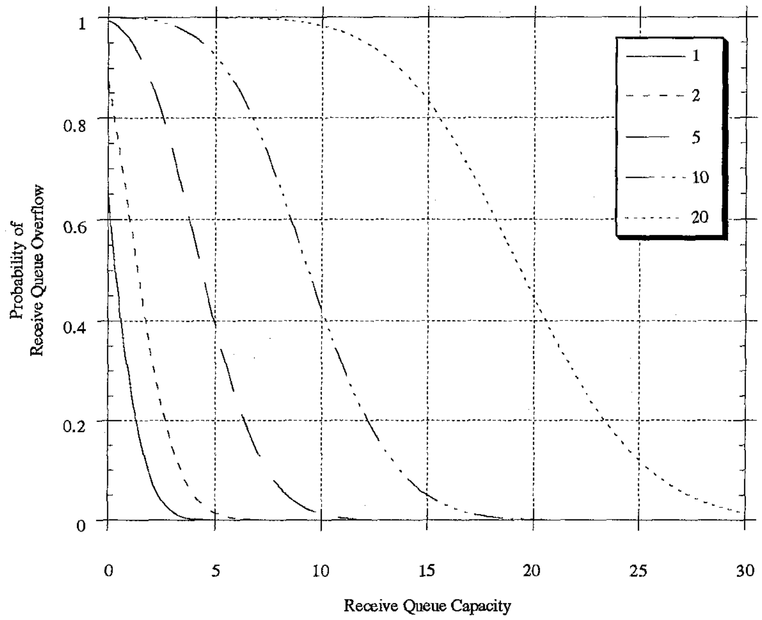


Figure 6.21: Probability of Receive-Queue Overflow Using Random-Random Placement. **The probability of receive-queue overflow is plotted as a function of the receive-queue capacity. Curves that represents the probability of receive-queue overflow during the concurrent placement of N, $2 \times N$, $5 \times N$, $10 \times N$, and $20 \times N$ processes are plotted. For each of these curves, the probability of receive-queue overflow is about 40% if the expected number of process-creation requests received by each node is equal to the receive-queue capacity.**

- Measure the memory utilization of the ensemble when a computation fails with and without message exportation. Preliminary results indicate that the computations frequently fail very early in the computation if messages cannot be exported, so that a tiny fraction of the machine is utilized when the computation fails.

- Monitor the number of messages that are exported as a function of (x, y). This experiment should illustrate message "hot-spots" of the computation.

- Vary the algorithms used to place exported-message remote processes (section 5.2.3). Increasing the distance that remote-process **new** messages travel increases the dissipation of "hot-spots".

- Monitor the number of times a message is exported and retrieved to ensure that the message-exportation algorithm does not introduce needless overhead.

Anticipated experiments with message exportation also include the simulation of ensembles that include nodes with special capabilities. For example, is the performance of message exportation significantly improved if the ensemble includes nodes that have a large available memory?

6.4.3 Code Management

The test-suite programs are, in effect, the inputs to experiments concerning the code-management strategies presented in section 5.2.3. The test suite already includes programs that, when split into code pieces, have a variety of numbers and sizes of code pieces, but additional programs should be added to extend this range.

Experiments with the code-management strategies in the MADRE system are expected to include:

- Vary the algorithms used for the initial placement of code pieces on ensemble nodes. Deterministic algorithms, *eg* even-numbered code pieces reside on even-numbered nodes, and randomized algorithms will be explored.

- Measure the performance of the Least-Recently-Used and Remote-Code-Execution code-management algorithms. Compare the wall-clock measurements of time required to the execute programs using each management algorithm.

- Evaluate the performance of hybrid code-management algorithms. For example, when the machine is lightly loaded, the code handlers should retrieve code pieces using the LRU strategy. As the machine becomes congested, the code handlers on affected nodes could begin using the RCE algorithm. Using a different hybrid algorithm, code handlers could compare the size of the desired code piece with the size of the state of the requesting user process. If the code piece is larger, the process state should be copied to the remote node as per the RCE algorithm. If the process state is larger, the code piece should be retrieved from its remote location using the LRU algorithm.

7 Conclusions and Future Work

By developing prototypes at each level of computation, our research group has created a framework for investigations into multicomputer software and hardware. In addition, since each of these prototypes has been developed with high-performance goals, the end result is an integrated system of practical value in solving application problems.

The Mosaic C has proved to be a versatile, reliable hardware platform for experimenting with system-level programming and higher-level user programming. The results presented in Chapter 6 illustrate the execution of C+- programs, supported by the MADRE runtime system. Our research group is currently working on the next version of the Mosaic architecture. This new prototype, called the Mosaic T, is a two-level multicomputer, where the first level handles message passing and the second level the execution of the user computation. The design of the C+- programming system and the MADRE runtime system have been guided by the eventual goal of separating message-passing function from execution of the user computation.

Using C+- to write the Mosaic runtime-system program has illustrated that C+- is a powerful programming tool. The expressivity of this notation is higher than previous multicomputer programming systems, without sacrificing execution efficiency. Future work on C+- includes the final implementation of process layering and refining the heterogeneous-machines interface mechanisms. An intense round of application-writing will follow to develop further C+- and object-oriented fine-grain multicomputer programming techniques.

The contributions of this thesis have centered on the MADRE runtime system. The design philosophy and structure of the MADRE system illustrate how a fine-grain runtime system can be efficiently distributed. The modular design of the MADRE system facilitates the tailoring of a runtime system to the target ensemble and the spectrum of expected application programs. By building an effective fine-grain runtime system, we demonstrate that the fine-grain architecture can exploit the concurrency in application programs.

This thesis also presents several effective runtime-system algorithms. The process-placement algorithms in Chapter 6 illustrate how easily algorithms can be interchanged in a modular runtime-system design. The process-placement performance results demonstrate that a runtime system can use a range of placement algorithms that trade off locality and machine utilization. Future work will include investigations of algorithms that rely on limited knowledge of the history of previous placement attempts. Each of process-placement algorithms also needs to be evaluated during execution of more sophisticated user programs. Executing programs on larger meshes is necessary to evaluate the practical effects of mesh boundaries on placement algorithms. A significant number of experiments also need to be conducted to evaluate the performance of algorithms used in MADRE for user-code management and ensuring robust execution of user programs (section 6.4).

Future work on the MADRE system will likely include the optimization of key runtime-system operations, such as computing DXDY, allocating buffers for incoming messages, and user-process stack management. The remote-process creation mechanism will probably be made available to C+− programmers. Given the proper mechanism for naming processes at the parent node, programmers can build process structures and refer to their constituent processes by computing reference values. The host program will become more sophisticated, including the capability for concurrent I/O to exploit the I/O bandwidth at the edges of a Mosaic mesh.

Bibliography

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1987.

[2] R. Arlauskas, P. Close, and S.F. Nugent. Assorted iPSC papers. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 38–60 (also 843–846), New York, 1988. ACM Press.

[3] W.C. Athas and C.L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, 21(8):9–24, 1988.

[4] William C. Athas. *Fine Grain Concurrent Computations.* Ph.D. thesis, California Institute of Technology, 1987.

[5] William C. Athas and C.L. Seitz. Cantor User Report Version 2.0. Computer Science Department 5232:TR:86, California Institute of Technology, 1986.

[6] Sandeep Bhatt and Jin-Yi Cai. Take a walk, grow a tree. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988.

[7] Peter Bickel and Kjell Doksum. *Mathematical Statistics: Basic Ideas and Selected Topics.* Holden-Day, San Francisco, CA, 1977.

[8] Nanette J. Boden. A Study of Fine-Grain Programming Using Cantor. Computer Science Department Caltech-CS-TR-88-11, California Institute of Technology, 1988. Master's thesis.

[9] Nanette J. Boden, Charles L. Seitz, Jakov Seizovic, and Wen-king Su. The design of the Caltech Mosaic C multicomputer. In Gaetano Boriello, editor, *Advanced Research in VLSI.* UW Press, to be published in 1993.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[11] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Boston, MA, 1992.

[12] Andrew Chien and William J. Dally. CST: An object-oriented concurrent language. *SIGPLAN Notices*, April 1989.

[13] William J. Dally. *A VLSI architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Norwell, MA, 1987.

[14] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. Technical Report EWD687, Phillips Research Laboratories, 1978. Eindhoven, The Netherlands.

[15] D.J. Kuck et al. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on the Principles of Programming Languages*, Jan. 1981.

[16] C.M. Flaig. VLSI Mesh Routing Systems. Computer Science Department 5241:TR:87, California Institute of Technology, 1988. Master's thesis.

[17] A. Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley Publishing Company, Reading, MA, 1991.

[18] *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, New York, 1988. ACM Press.

[19] Arjun Khanna. On managing classes in a distributed object-oriented operating system. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990.

[20] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1973.

[21] Charles R. Lang, Jr. *The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture.* Ph.D. thesis, California Institute of Technology, 1982.

[22] S.L. Lillevik. DELTA:a 30 Gigaflop Parallel Supercomputer for Touchstone. In *NORTHCON Conference Proceedings*, pages 294–304, 1990.

[23] Johan J. Lukkien and Jan L.A. van de Snepscheut. A tutorial introduction to mosaic pascal. Computer Science Department Caltech-CS-TR-91-02, California Institute of Technology, 1991.

[24] John Y. Ngai. *A Framework for Adaptive Routing in Multicomputer Networks.* Computer science department, California Institute of Technology, 1989.

[25] Douglas M. Pase and Allan R. Larrabee. Intel iPSC Concurrent Computer. In Robert G. Babb II, editor, *Programming Parallel Processors.* Addison-Wesley Publishing Company, Reading, MA, 1988.

[26] P. Pierce. The NX/2 Operating System. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, New York, 1988. ACM Press.

[27] Charles L. Seitz. Mosaic C: An experimental fine-grain multicomputer. In *Proceedings of the Twenty-Fifth Conference INRIA Conference*, New York, 1992. Springer-Verlag Press.

[28] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, 1985.

[29] C.L. Seitz. Concurrent architectures. In *VLSI and Parallel Computation.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[30] C.L. Seitz. Multicomputers. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[31] C.L. Seitz, W.C. Athas, C.M. Flaig, A.J. Martin, J. Seizovic, C.S. Steele, and W.-K. Su. The architecture and programming of the Ametek Series 2010 multicomputer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, New York, 1988. ACM Press.

[32] C.L. Seitz, J. Seizovic, and W.-K. Su. The C programmer's abbreviated guide to multicomputer programming. Computer Science Department Caltech-CS-TR-88-1, California Institute of Technology, 1988. (revised 1989).

[33] Jakov Seizovic. The Reactive Kernel. Computer Science Department Caltech-CS-TR-88-10, California Institute of Technology, 1988. Master's thesis.

[34] Jakov Seizovic. *Mosaic and the C+− Programming Notation.* Ph.D. thesis, California Institute of Technology, 1993.

[35] Don Speck. Mosaic RAM design. In William J. Dally, editor, *Advanced Research in VLSI.* MIT Press, 1991.

[36] Craig S. Steele. *Affinity, A Concurrent Programming System for Multicomputers.* Ph.D. thesis, California Institute of Technology, 1992.

[37] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1991.

[38] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[39] Wen-king Su. *Reactive-Process Programming and Distributed Discrete-Event Simulation.* Ph.D. thesis, California Institute of Technology, 1989.

[40] Wen-king Su, Reese Faucette, and C.L. Seitz. C programmer's guide to the COSMIC CUBE. Computer Science Department 5203:TR:85, California Institute of Technology, 1985. (Out of Print).