A Message-Driven Programming System for Fine-Grain Multicomputers [1]

Daniel Maskit
Scalable Concurrent Programming Laboratory
California Institute of Technology
Pasadena, California

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

February 1, 1994

[1] The research described in this report is sponsored primarily by the Advanced Research Projects Agency, ARPA Order number 8176, and monitored by the Office of Navel Research under contract number N00014-91-J-1986. The author is partially supported by an NSF Graduate Research Fellowship.

Acknowledgments

This work owes a great debt to other members of the research groups at both the California Institute of Technology and the Massachusetts Institute of Technology. First among these is my thesis advisor, Steve Taylor, who has always had encouraging words and guidance when I needed them most. In addition, Yair Zadik and Chris Ziomkowski implemented the linker, loader, archiver, and floating-point support. Dong Lin collaborated on the initial version of the compiler. Andy Fyfe provided invaluable assistance with the C libraries and floating-point routines. Andrew Chang, Mike Noakes, and other members of the Concurrent VLSI Architecture project at MIT have provided constant assistance with low-level software and hardware.

Contents

Chapter 1

Introduction

Recently, two radical new architectural experiments have been conducted at Caltech and MIT. At Caltech, the Mosaic architecture, developed by the Submicron Systems Architecture Project, is designed expressly to support efficient fine-grain process execution [24]. The J-machine is a similar design, developed at MIT by the Concurrent VLSI Architecture Project, that supports fine-grain processes but also provides on-chip associative memory, and hardware support for process synchronization [10].

One of the contributions of these fine-grained architectures is that they have changed the basic set of assumptions for parallel computation. Traditional parallel machines have supported local computation at a significantly lower cost than communication. The result of this has been languages and programs that emphasize executing large pieces of code with minimal communication. Previous reactive programming systems [4, 7, 24] hide latency by overlapping computation and communication, but strive to minimize communication. Hardware platforms such as the J-Machine allow the programming system to do far more communication than earlier machines such as the Intel Touchstone Delta. On the J-Machine the latency of fetching data from local memory is comparable to sending that same piece of data to another computer within the machine for processing. This change in the equation of parallel execution presents new challenges to the compiler writer.

The most obvious new challenge posed by fine-grained machines is that of developing programming systems that efficiently use the specialized hardware of the machine for performing communication and synchronization, without complicating the programmer's task of specifying concurrency.

This thesis describes an experimental message-driven programming system and its implementation on a 512-computer J-machine [10]. This machine is an architectural experiment which focuses on the evaluation of hardware mechanisms, such as the integration of messages and processes, to support concurrent programming. The machine combines a unique collection of architectural features that include *fine-grain processes*, *on-chip associative memory*, and *hardware support for process synchronization*. The programming system utilizes these mechanisms via a simple message-driven process model that blurs the distinction between processes and messages: messages correspond to processes that are executed elsewhere in the network. This model allows code and data to be distributed across the computers in the machine, and is supported at every stage of the program development cycle. Although the concepts are language independent, the prototype system is based on GNU-C.

The programming system carries the experience gained from previous experiments [12, 26, 14] into a C-based system, while exploiting the special features available in the underlying architecture. The basic programming model for the system is:

> A computation is a collection of concurrent processes that may execute in any order or in parallel. Processes communicate and synchronize using shared variables. Mapping is achieved using annotations, for example *foo(...)@n* which specifies that process *foo(..)* is executed at computer *n*. Processes allocate and deallocate memory piecemeal when necessary and do not employ a stack; they may share global variables. A process may *suspend* at any time during its execution for the purpose of covering latency while relocating code or data.

The primary goals for the implementation were to support message sending and receiving, as well as process suspension and resumption, without having to perform copying. Since the limited amount of memory available at each computer (1/4 megaword) is less than that required for the code of some of the applications under development in the research group, a code distribution scheme was necessary from the outset. The resulting programming system employs a novel implementation strategy for fine-grain programs that has the following characteristics:

- New architectural features are directly accessed through native code compilation.
- Hardware performance is delivered directly to applications by removing software overheads associated with message-passing.
- Code and data distribution are provided by a simple run-time microkernel.
- Communication latency is hidden by a process suspension mechanism.
- Processes utilize a heap-based allocation scheme rather than a stack and may thus suspend without copying overheads.

This thesis describes the programming system and experiences with its development and use. The prototype system includes a compiler, linker, archiver, loader and microkernel. It is currently being used for a variety of large-scale applications experiments in computational fluid dynamics, circuit simulation, and molecular modeling. Although the examples presented in this work are of small programs, they represent all of the issues for concurrency that exist in the larger applications being developed by the group.

Chapter 2

Related Research

The basic concept of message-driven process execution was integrated in Chuck Seitz's hardware design philosophy for multicomputer architectures, and incorporated in a variety of reactive programming systems [4, 24, 25]. This concept has been a recurrent theme in both hardware and software designs from the Submicron Systems Architecture Project over the years. It has been incorporated directly into most commercial multicomputer programming systems. In common with Cantor and Actor [1] programs, this work utilizes message-driven concurrent processes that do not employ a stack. However, unlike these systems, *Message-Driven C* processes typically communicate and synchronize via a simple shared variable concept, suspend to cover latency, and may share global state for efficiency. It is important to recognize that although global variables are supported, the intent is to capitalize on their efficiency when used in a good software engineering style for building abstract data types [11]. Their use for global sharing between program components in a completely unstructured manner such as that found in most FORTRAN programs is discouraged. An example of an MDC program that computes $n!$ can be found in Figure 2.1. Note the use of the mapping anotation @ to specify the location for execution of each recursive call.

Recently, a new generation of fine-grain systems have appeared. The MADRE [5] system developed for the Mosaic Architecture [24] distributes the microkernel across multiple computers, and hides the details of process to processor mapping from the user. The major emphasis of MADRE is to provide a system that handles resource allocation, and is capable of using resources across the entire machine if a particular computer exhausts its local resources. In contrast, the philosophy of the work presented in this thesis is that the programmer is best able to make decisions about process mapping, and, furthermore, that this decision-making is simple. If a program exhausts local resources, then it is considered to be a poorly-formed program. It is left to the programmer to restructure the code to better balance its resource usage.

The language that is supported on the Mosaic for use with the MADRE runtime system is *C+-* [23]. This language is a superset of *C++*, with additions for process creation and destruction, and communication. The particular emphasis of this work is on supporting

```
int factorial(int n)
{int answer;
    if(!n)
            answer = 1;
    else {
            int i = factorial(n-1)@next;
            answer = n * i;
    }
    return answer;
}
```

Figure 2.1: Factorial Program in MDC

the *reactive* programming model. Figure 2.2 shows an example of a C+- program. Note the absence of mapping annotations. While the language does allow the programmer to specify mappings, it's authors recommend that this process be left to the runtime system.

```
processdef factorial
{   public:
    atomic int compute(int n) {
        if(!n)
                return 1;
        else {  factorial f1;
                return (n * f1.compute(n-1));
        }
    }
}
```

Figure 2.2: Factorial Program in C+-

A Concurrent Smalltalk (CST) [16] system has also been developed for programming the J-machine. CST requires a larger and more complex microkernel to handle a broad range of language concepts that applications within the *Scalable Concurrent Programming Laboratory* do not require. Figure 2.3 shows an example of a CST program. Note that CST provides a mapping annotation very similar to that used by MDC. The *Message-Driven C* system is currently the only programming system that has been developed for the J-machine that is broadly considered a useful platform for large-scale scientific applications.

```
(Defmethod factorial Integer ():Integer
    (   if (= self 0)
              1
        (* self (factorial @(next) (- self 1)))
    )
)
```

Figure 2.3: Factorial Program in CST

The *Scalable Concurrent Programming* group has been involved for several years in developing portable, high-performance programming systems that execute efficiently on scalable multicomputers [24]. The high-level programming systems that have been developed within this group [12, 26, 14], all share the same fine-grain process model present in this work. An example program written in one of these languages, PCN, can be found in Figure 2.4. The mapping annotations used by PCN and MDC are identical. Unfortunately, scalable multicomputer architectures have traditionally supported only Unix-style, coarse-grain, stack-based processes. Thus previous implementations have been forced to utilize an emulation technique to provide efficient systems [13, 26]. The J-machine supports fine-grain processes and also provides on-chip associative memory, and hardware support for process synchronization [10]. The programming system described here provides a low-level platform that supports both irregular applications and development of high-level systems on fine-grain multicomputers.

```
factorial(y, result)
{   ? y == 0 → result = 1,
    y > 0 → {|| result = y*r1, factorial(y-1, r1)@next}
}
```

Figure 2.4: Factorial Program in PCN

The idea of using a heap-based implementation of a stack-oriented language was earlier reported for Pascal by Marlin [18]. This work focused primarily on the Pascal-specific issues of the implementation. In *Message-Driven C*, heap-based frames are used to support light-weight processes. Although Marlin's work was discussed as a predecessor to a system that supports multi-threaded execution via coroutines, this work shows that a variety of means for expressing concurrency can be obtained through trivial extensions to the heap-based system.

There have been a number of alternative approaches to the issues presented here. Arvind and Nikhil at MIT have argued that problems with supporting parallelism on a sequential architecture machine relate to intolerance to the high memory latency required for concurrent programs and the lack of acceptably fast synchronization mechanisms [3]. These ideas led to the development of the Tagged-Token Dataflow architecture, a machine that uses additional bits on each data word for synchronization, and is designed to provide optimal support for dataflow programs. Arvind and Nikhil argued that while one could run a parallel version of a sequential language on this machine, this complicates compilation significantly, and will not give acceptable runtime performance. Given the appropriate hardware support, it is clear that efficient parallel implementations of sequential languages can be constructed. In addition, in earlier publications from the *Scalable Concurrent Programming Laboratory* [14], the feasibility of *layering* concurrency abstractions on top of a system that provides only the basic support described in this thesis has been demonstrated. This can be achieved through the use of source-to-source transformations.

Schauser, Culler and von Eicken have developed the idea of compiler decomposition of a program into threads, some of which can be statically scheduled [22]. The motivation for this work is the belief that high-speed context switching in hardware is hard. The philosophy of this work is that it is best to leave control of program decomposition to the user, or to tools specific to a class of applications. The hardware process support offered by the J-Machine provides acceptable context-switching performance.

Although there have been a variety of systems that have implemented concurrent versions of the C language, such as [6, 15], the previous systems that have involved multicomputer implementations have focused on developing a system that relies on UNIX-style support for system services. The emphasis of this work is on utilizing specialized multicomputer hardware mechanisms to support process creation, communication and suspension, thereby minimizing the amount of time that is spent running system code. In addition, the programming system described here is usable for large-scale applications, rather than simply providing a minimal system to demonstrate the validity of the ideas.

Gul Agha, in recent work at the University of Illinois at Urbana-Champaign has been working on expanding the Actor model to better reflect the realities of concurrent programming. In [2] he discusses the use of Actors within an object-oriented programming framework. The rationale behind this type of framework is the need to provide the programmer with support for abstraction. In particular, the concern of managing concurrency without unduly restricting the expressiveness of the source language is addressed. The work also addresses the difficulties of adequately and efficiently supporting concurrency using an Object-Oriented paradigm. One of the major areas of concern explored is the problem with locality of objects, and how poor placement or construction of objects can have an adverse effect on efficiency of the system. One of the approaches to solving this problem is the use of program transformations to increase locality. An example of an Actors program, reproduced from [1], can be found in Figure 2.5. As Actors is intended to provide an abstraction of concurrency which emphasizes processes without reference to processors, it is not necessary to provide annotations for mapping.

```
def exp Rec-Factorial()[n]
    become Rec-Factorial()
    if n = 0
        then reply [1]
        else reply [n * (call self[n-1])]
    fi
end def
```

Figure 2.5: Factorial Program in Actors

Andrew Chien, in similar work also done at the University of Illinois at Urbana-Champaign explores the use of *Concurrent Aggregates* to provide an object-oriented approach to implementing a concurrent version of Actors [8]. The difference between this work and Agha's work is that Chien is experimenting with a new variation on Actors which he calls *aggregates*. An *aggregate* is a collection of Actors in which each of the Actors can concurrently receive messages. This structure allows a relaxation on the ordering of message reception present within the Actor paradigm. This is useful as it provides the ability to support the core ideas of the Actors paradigm within the inherently non-deterministic framework of a message-passing multicomputer, without incurring overhead costs to handle messages in an order dictated by the software abstraction.

Both projects working with Actors are complementary to the work described in this thesis. One of the goals of the message-driven C system is to support concurrency without needlessly complicating the source language. Greater support for abstraction, as well as the use of transformations to improve communication performance, are both topics considered as future work.

Chien is also involved in exploring optimization techniques for concurrent object-oriented programs executing on multicomputers constructed with stock processors, such as the CM5 [27]. This work [9] describes the design of the **Concert** system. The key idea in this system is the performance of optimizations at all points in the program compilation and execution cycle. Early optimization is performed using source-to-source transformations in the front-end of the compiler. When there is not enough information to statically determine the most efficient form of a program, multiple copies are produced, and the runtime system selects between the different versions, attempting to select the lowest-cost variation for each invocation. Inasmuch as this system is targeted at commodity processors, and relies on a substantial runtime system, it is philosophically quite different from message-driven C. However, the work in static optimization is closely related to Gul Agha's work, and is similarly relevant to future research directions.

Chapter 3

System Design

This chapter describes the design of the programming system. The design is motivated first by describing the programming model in use, then by providing an abstract view of the hardware as seen by the programming system. The remainder of the chapter explains how the programming system bridges the gap between these two abstractions, concluding with an example program which illustrates this bridging.

3.1 Programming Model

Recall that the computational model involves a collection of concurrent processes that may execute at any computer. A process is a sequential locus of control defined using the C language. To support this model, a hardware abstraction of a fully connected set of computers numbered from *0* to $n-1$ is provided. A simple communication library is used to provide process mapping; it consists of just three functions:

> **computers().** Returns the number of computers in the allocated machine.
>
> **computer().** Returns the identifier of the current computer, an integer in the range 0 to $computers() - 1$.
>
> **x = f(l,p)@n.** Spawns the function named f as a concurrent process on computer n with two arguments and returns value x. p is a pointer to program data and l is the corresponding data length.

3.2 Abstract View of the J-Machine

This section provides an abstract description of the J-Machine hardware. The machine provides three distinct hardware mechanisms to support concurrent programming: message queueing and dispatch; a hardware *associative cache*; and support for synchronization. Figure 3.1 provides an abstract view of the hardware that highlights these features.

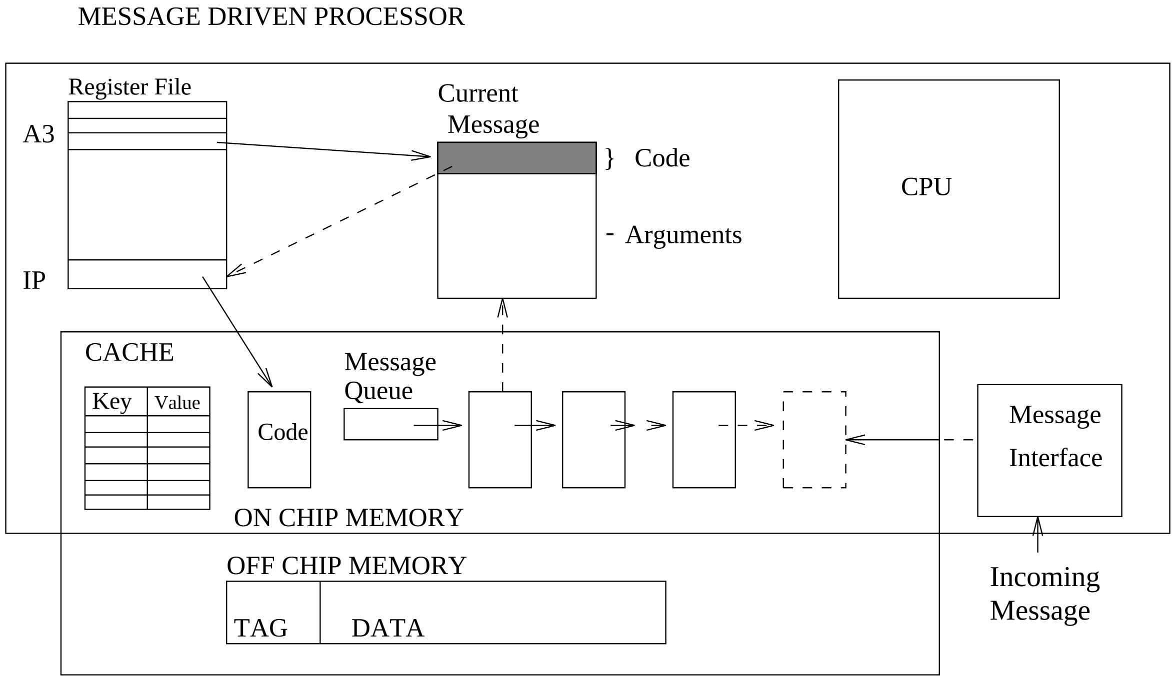

Figure 3.1: An Abstract View of the J-Machine

A message corresponds to a process in transit. Each message carries the address of the code to execute and the process arguments. Upon arrival, each message is appended to a hardware message queue that contains all active processes. A process is executed by removing it from the message queue, placing its code address into the instruction pointer, and placing a pointer to its arguments in machine register A3. This message dispatch is fully implemented in hardware. The process then begins execution with its arguments residing in the original message, accessed via register A3.

The machine provides an associative memory that allows a full 36-bit association between a key and its value. This memory can be manipulated via simple *enter*, *translate*, and *probe* operations. The *enter* operation adds an entry to the table at some index. The *translate* operation reads the data associated with an index. Finally, the *probe* operation inspects the table to determine if a location is in use. These functions allow the associative memory to be used as a hash table.

A final feature that is useful is a 4-bit *tag* field associated with every memory word. This can be used to provide a simple process synchronization mechanism. The tag may be preset to an *undefined* state. If a memory word is accessed while its tag is set to *undefined*, for example during an arithmetic operation, an exception is generated and a fault handler executed.

3.3 The Programming System

A process is defined by sequential C code, however remote processes can communicate and synchronize using shared variables, which are implemented using hardware *tags*. Mapping is achieved using annotations, for example *foo(...)@n* which specifies that process *foo(..)* is executed at computer n. This results in the sending of a message to computer n, requesting that it locate and execute function *foo* using the arguments present in the message. Processes allocate and deallocate memory piecemeal when necessary and do not employ a stack; they may share global variables. A process may *suspend* at any time during its execution for the purpose of covering latency while relocating code or data.

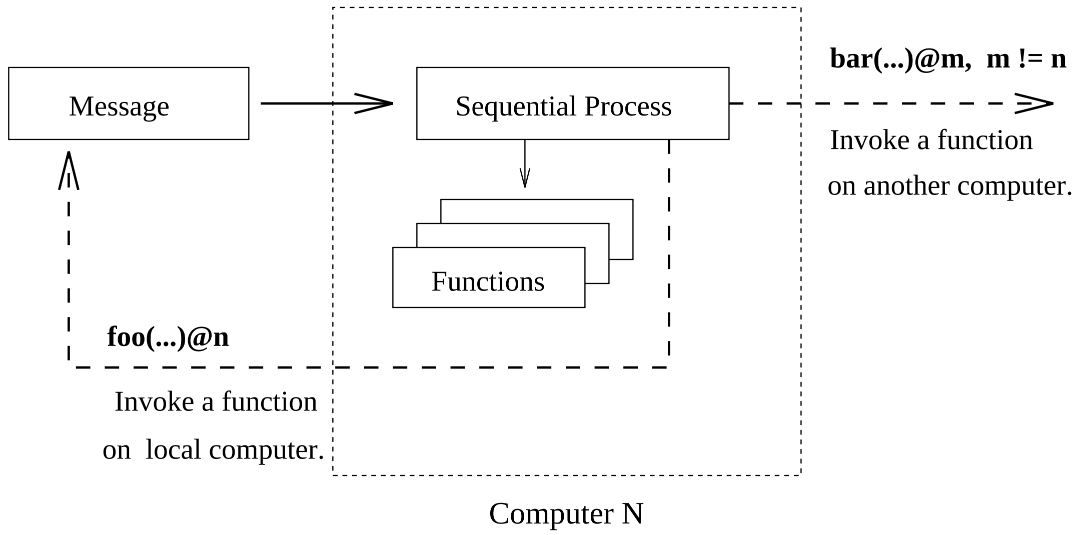

Figure 3.2: Single Computer Operation

Figure 3.2 outlines the use of the communication library functions for constructing low-level programs. Remote functions execute as independent concurrent processes and may invoke other functions at the current computer during their execution. They may also cause new processes to be created at any computer in the machine using the *spawn* notation *@n*. Attempting to access any part of return value x prior to its being available will cause the calling process to suspend. If function f is a void function there is no synchronization involved: the calling function proceeds immediately without waiting for termination of the remote function. Notice that the mechanism for remote process invocation follows the process model provided by the hardware. This allows the hardware to be directly used by the implementation.

3.3.1 Example Low-Level Program

Figure 3.3 shows a simple example of the use of the low-level programming system. The program begins execution at the keyword *main* on computer zero. It then traverses the entire machine one computer at a time in numerical order. At each computer the program prints the computer number multiplied by the number of times that computer has been visited. Thus on a multicomputer with three processors, the output would be:

0 1 2 0 2 4 0 3 6 0 4 8

```
#define NUM_REPS 4

main()
{   int rep = 0;
    Hop(sizeof(rep), &rep);                            /* Local call */
}

void Hop(int size, int* rep)
{   int next = (computer()+1) % computers();  /* Where next */
    if(computer() == 0) (*rep)++;
    if(*rep <= NUM_REPS) {                              /* Not done ? */
        printf("%d ",computer()*(*rep));              /* Print computer number */
        Hop(sizeof(int),rep)@next;                      /* Hop on */
    }
}
```

Figure 3.3: Example Low-Level Program

Chapter 4

Implementation Details

This chapter describes the implementation of the programming system. In outline, the programming model involves a collection of concurrent processes that may be transported between processors and may suspend while waiting for code or data. These processes are implemented as J-Machine messages that are scheduled directly by the hardware. From the programmer's perspective, this model is supported by the simple communication library discussed in Section 3.1. The code for the processes may be distributed across the entire machine. In addition to retargetting the GNU C compiler to support this scheme, new linker, loader and archiver tools were developed. These tools operate in conjunction with a small run-time microkernel. The microkernel, which is approximately 200 words in length, uses the hardware *associative cache* to keep track of what code is present at a single computer. Process synchronization is achieved by manipulating *tag bits* associated with data in transit. Thus all of the main hardware mechanisms are utilized by the system.

4.1 Compiler

To support the basic programming model the GCC compiler was retargetted to generate sequential C code for the J-Machine. The primary motivations for choosing GCC were the desire to take advantage of the significant amount of optimization that is performed by the GNU system, the desire to provide a complete and integrated collection of system libraries, and the proven track record of GCC for being ported to diverse platforms.

4.1.1 Code Generation

Figure 4.1 shows a sample piece of C code that implements a typical producer process. As stated, this routine will send 100,000 messages to computer 8 from wherever it is running. Each of these messages will result in the creation of a process that will run the function **consumer** with access to a copy of the array a. The compiler output for the example code is shown in Figure 4.2 and Figure 4.3. All of the instructions which are required have comments next to them describing their purpose. Only 6 out of 108 instructions, or 5.56%, are unnecessary. Significant time has not been invested in developing peephole optimizations for the current revision of the compiler. As the compiler matures it is expected that the number of unnecessary instructions generated will decrease.

```
#define ITERATIONS 1000000                     /* Number of messages to send */
#define TARGET 8                               /* Computer to send messages to */

void producer(void)
{   int i;
    int a[8] = {0, 1, 2, 3, 4, 5, 6, 7};       /* Data to Send */

    for(i = 0; i < ITERATIONS ; i++)           /* Send ITERATION messages */
        consumer(sizeof(a), a)@TARGET;
}
```

Figure 4.1: Code Generation Example

In terms of communication and synchronization, the generated code has the following main components (the reference numbers indicate locations in Figure 4.2 and Figure 4.3):

Process mapping. The physical address of the target node, designated with the annotation *@TARGET*, results in the generation of the physical address of node 8 **(1)**.

Direct compilation. The use of the mapping anotation @ requests the sending of a message. As can be seen **(2)** this compiles directly into the use of hardware **send** instructions.

Send without copying. As can be seen in between **(2)** and **(3)** the message data is sent directly out of memory without any need to copy it into an intermediate message buffer first.

```
_producer::
      move 14, r3            ; Size of local variables
      move 0, r1             ; Size of outgoing arguments
      CALL make_frame        ; Create call frame on heap
      dc INT:(LC0)           ; Define Constant: Global address of initialization array
      move [r0,a0], r1       ; Get 0th element of initialization array
      move -14, r0           ; Generate offset for local variable
      move r1, [r0, a1]      ; Initialize 0th element of array
      ⋮                      ; 28 Instructions to initialize the rest of array
      move 0, r1             ; Move 0 into register R1
      move -15, r0           ; Generate offset for local variable
      move r1, [r0, a1]      ; Initialize loop counter
(1)   dc INT:(264)           ; Define Constant: Pointer to computer address table
      move [r0,a0], r3       ; Load pointer to computer address table
      add r3, 8, r0          ; Add target computer logical number to base of table
      move [r0,a0], r3       ; Load physical address of target computer from table
      move -17, r0           ; Generate offset for local variable
      move r3, [r0, a1]      ; Store physical address of target computer
L54:  move 1, r0             ; Move 1 into register R0
      move r0, I             ; Disable interrupts
      move -17, r0           ; Generate offset for local variable
      move [r0, a1], r1      ; Load physical address of target computer
(2)   send r1,0              ; Start send with physical address of target computer
      move -16, r0
      move [r0, a1], r3
      dc INT:$80200010       ; Define Constant: Message Header
      move r0, r1            ; Move into general use register
      wtag r1,MSG,r3         ; Tag word as Message. Required by hardware
      send r3,0              ; Send message header
      move -16, r0
      move r3, [r0, a1]
      dc INT:_consumer       ; Define Constant: Function identifier for "consumer"
      move r0, r3            ; Move into general use register
      send r3,0              ; Send identifier of function to invoke at remote computer
      send 0,0               ; Send blank word to be used as temporary storage at receiver
      send -1,0              ; Send -1 to indicate that no return value is necessary
      send 0,0               ; Field indicating which computer expects return value
      send 0,0               ; Field indicating memory address for return value
      move 1, r1             ; Move 1 into register R1
      send r1,0              ; Send number of arguments for remote function
```

Figure 4.2: Prologue through transmission of argument count

```
        move 8, r3              ; Move 8 into register R3
        send r3,0               ; Send length of argument for remote function
        move 0, r2              ; Initialize loop counter
        move a1, r1             ; Move address register A1 into register R1
        wtag r1, INT, r1        ; Change tag on R1 to INTeger
        ash r1, -10, r1         ; Shift out size field from pointer
        move r1, r1
        add r1, -5, r1          ; Add -5 to R1, Now contains address of last item in array+ 2
        move -18, r0            ; Generate local variable offset
        move r1, [r0, a1]       ; Store R1 to local variable
L46:    move -18, r0            ; Generate local variable offset
        move [r0, a1], r3       ; Load address of end of array + 2
        add r3, r2, r3          ; Add loop counter
        add r3, -9, r0          ; Generate address of next element in array to send
        move [r0,a0], r1        ; Load array data
        send r1,0               ; Send array data
        add r2, 1, r2           ; Increment loop counter
        le r2, 6, r0            ; Have we sent all but one item yet?
        bf r0, ^L55             ; If we have, break out of loop
        dc L46 - (* + 2)        ; Generate address of L46 relative to Instruction Pointer
        br r0                   ; Jump to top of loop
L55:    move -7, r0             ; Generate local variable offset
        move [r0, a1], r3       ; Load last element in array
(3)     sende r3,0              ; Terminate send with last data item
        move 0, r0              ; Move 0 into Register R0
        move r0, I              ; Enable Interrupts
        move -15, r0            ; Generate offset for local variable
        move [r0, a1], r1       ; Load loop counter
        add r1, 1, r1           ; Increment loop counter
        move -15, r0            ; Generate offset for local variable
        move r1, [r0, a1]       ; Store loop counter
        dc INT:99999            ; Define Constant: ITERATIONS
        le r1, r0, r0           ; Compare counter to ITERATIONS
        bf r0, ^L56             ; If counter > ITERATIONS, branch to L56
        dc L54 - (* + 2)        ; Generate address of L54 relative to Instruction Pointer
        br r0                   ; Jump to top of loop
L56:    move [0,a2], r2
        move -1, r0             ; Move -1 into R0
        move [r0,a1], r1        ; Get address of current frame
        move [r0,a2], r0        ; Get address of previous frame
        CALL function_end       ; Call routine to deallocate current frame
        ldip r0                 ; Function Return
```

Figure 4.3: Transmission of argument length through Epilogue

4.2 System Tools

To support code distribution it was necessary to develop a collection of system tools, and to devise a scheme that would allow the essential system code to be resident at every computer. This was achieved by separating all program functions into two categories: *replicated* or *distributed*. Distributed functions are pieces of code that permanently reside at one computer (the function's *home computer*), but can be transported anywhere in the machine when they are needed. Replicated functions are pieces of code that exist at all computers. Figure 4.4 shows how these functions are generated in the compilation pipeline. The MDC compiler compiles files in the usual manner and generates object code (e.g. a.o, b.o and c.o). The main purpose of our object file format is to allow for all functions to be treated as independent units that can be mapped onto any computer and easily moved to any other computer. The linker combines these object files to form an indexable binary. Object code which is to be *replicated* is signified using a linker flag (*-r*). The resulting binary contains code for each computer, as well as *replicated* code and global variables that exist at all computers.

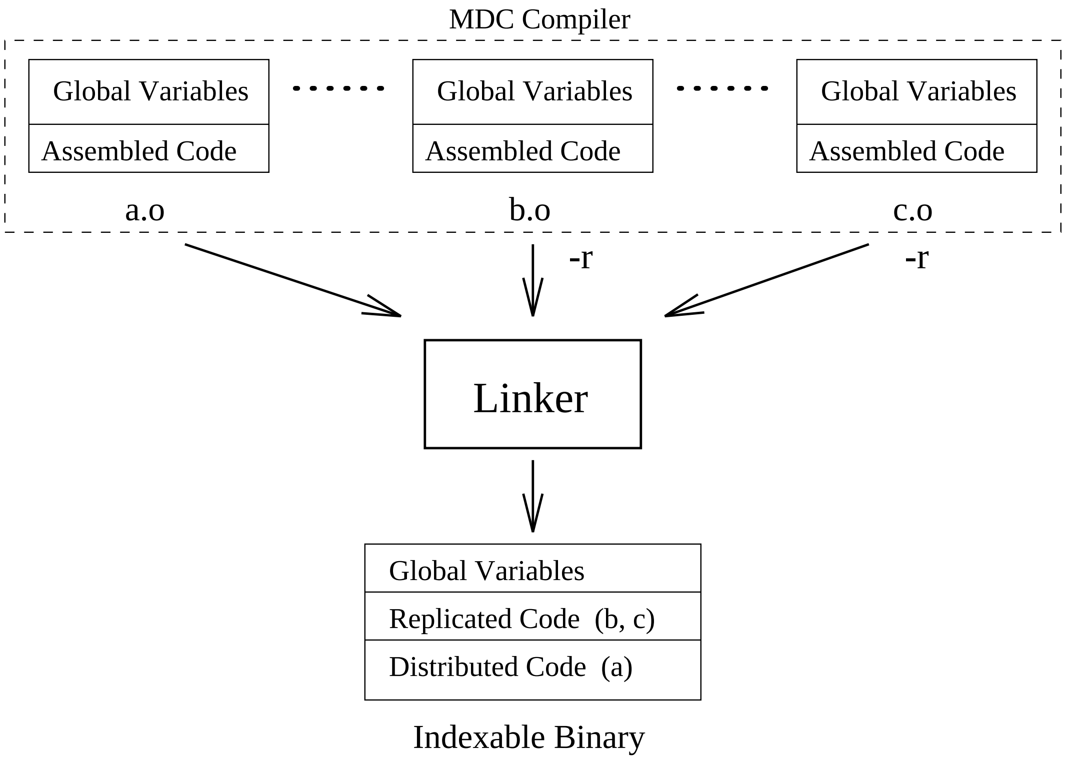

Figure 4.4: Linkage and Loading Stages

The linker assigns each distributed function to a *home computer* in the machine. A simple bin-packing algorithm is used to balance the quantity of code resident at each computer; there is a standard interface to the linker that allows experimentation with alternative code mapping algorithms. In the current implementation, global variables are replicated at every computer and coherence is not maintained between computers.

After code mapping is complete, the linker assigns unique identifiers to all functions. If a function is invoked during program execution, the microkernel uses these identifiers to locate the appropriate code. The identifier is a pair of the form:

$$< \textit{logical-computer}, \textit{function address} >.$$

The *logical-computer* entry signifies the home computer of the code, and the function address specifies the location of the code at the home computer. The translation from logical-computer to physical-computer is made by the loader after allocation of a partition of the machine.

4.2.1 Layers and Interfaces

The message-driven C system is composed of a number of different programs, with file formats that act as interfaces between the programs. The description of system tools above, combined with Figure 4.4, provide context for most of these interfaces. Although many of these interfaces are based on work done by other people, the final form of the interfaces in this system reflect special needs and goals particular to message-driven C.

The initial assembly language definition was developed by Waldemar Horwat at MIT [17]. This language was modified by this author as described in [20] to provide explicit support for code distribution, as well as improved efficiency of other tools in the system. These modifications included a separation between data and code to facilitate mapping them to different areas within the machine; the addition of assembler directives to provide a delineation of the beginning and end of functions to support modularization and code transport; and a new notation for specifying global variables which simplifies specification of uninitialized data regions, and provides size information for initialized data.

The original assembler output was a text file containing an ASCII representation of the assembled code. This output format required substantial work on the part of the programmer to guarantee that multiple files could be downloaded to the J-Machine and executed as one program. The assembler output was changed from this simple text file into a binary object file. The exact layout of this object format was designed by Yair Zadik, based on the changes made to the assembly language, and the design of the distributed code mechanism by this author. The format of object libraries is the standard BSD UNIX format supported by the standard utility program **ar**. The index for the library was designed by Yair Zadik.

The layout of the executable file produced by the linker was designed by Yair Zadik based on the needs of the runtime microkernel, which was designed by this author.

4.3 The Microkernel

The microkernel is the portion of the system that provides support at runtime for process creation, suspension and awakening, as well as code transport of distributed functions. The general organization of the microkernel is a collection of handlers that may be invoked at any time upon receipt of a message. The microkernel is notable for its size which is less than 200 words of memory; this is attained through the simplicity of the computational model. There are five primary handlers which make up the microkernel; these are shown in Figure 4.5.

```
SPAWN_HANDLER:
    if (function replicated OR this is home computer OR in hash table)
        invoke function
    else {
        if(not requested already)
            send REQUEST message
        suspend process in queue associated with function
    }

REQUEST_HANDLER:
    Send function to requesting process in RECEIVE message

RECEIVE_HANDLER:
    Allocate storage for function code
    Copy code from message to storage
    Enter function in hash table
    Wake processes waiting for function

SET_HANDLER:
    For each word to be written
        If memory not zero wake up suspended processes
        Write value into memory

WAKEUP_HANDLER:
    Load registers from state stored in process header
    Load instruction pointer with suspension address
```

Figure 4.5: Microkernel Message Handlers

Spawn. The *spawn* handler is responsible for locating the code of a function and subsequently executing it. Recall that this occurs when a program executes *f(...)@n*. The handler is used to manage both spawn messages, originating at other computers, and local function invocation. A function can be executed immediately if it is either replicated or it is distributed and the current computer is the home computer. Otherwise, the code is either in a local hash table or must be fetched from its home computer and deposited into the hash table. The hash table is implemented using the *associative memory* provided by the J-machine hardware. If code is requested from another computer, then the current process suspends until this code is received locally.

Request and Receive. The *request* handler deals with a request for code from another computer and transmits the code from a known location determined using the *function address* portion of the code's unique identifier. Finally, the *receive* handler is invoked when requested code is received. It wakes all processes that are suspended awaiting code reception and updates the local hash table to store the received code. The hash table is purged to free space when necessary.

Set and Wakeup. The microkernel also has two routines to support the setting of shared variables being used for synchronization, and the awakening of suspended processes. Recall that when a process executes x = *f(...)@n*, it will suspend if x is accessed before it is available. The *set* handler accepts a message containing an address and length, as well as *length* words of data to write starting at the specified address. Before writing each word, this handler checks to see whether or not the present value is 0. If it is non-zero, than the word contains the address of a list of suspended processes. For each process on this list, a message is sent to the *wakeup* handler. The *wakeup* handler accepts messages that contain the address of the process header data structure of a suspended process. This handler loads the machine registers with the state stored in the process header, and loads the instruction pointer with the address of the instruction that caused the process to be suspended.

4.4 Remote Function Invocations

The message-driven process model of the hardware is utilized directly for remote function invocation, using *f(...)@n*. The remote function calls are compiled directly into the sending of a message which transmits data from memory directly into the message-passing network; thus, there are no copying overheads associated with message sending. Message reception results in the creation of a process record which includes the contents of the received message. The copy from the message buffer into the process record is the only copying of data that is performed by the system. Since the message data resides in the process record, it is persistent across suspension without the kernel having to take any action to preserve it. This persistence has been extended by placing control of deallocation of arguments with the application program. This allows an application to store a received message in persistent variables for later use without additional copying. By default a process record, and the message data contained within it, is never destroyed.

When a program completes operations on the data contained within a specific message, a *free* operation is used to deallocate the memory used by the process record that contains that message.

4.5 Process Suspension

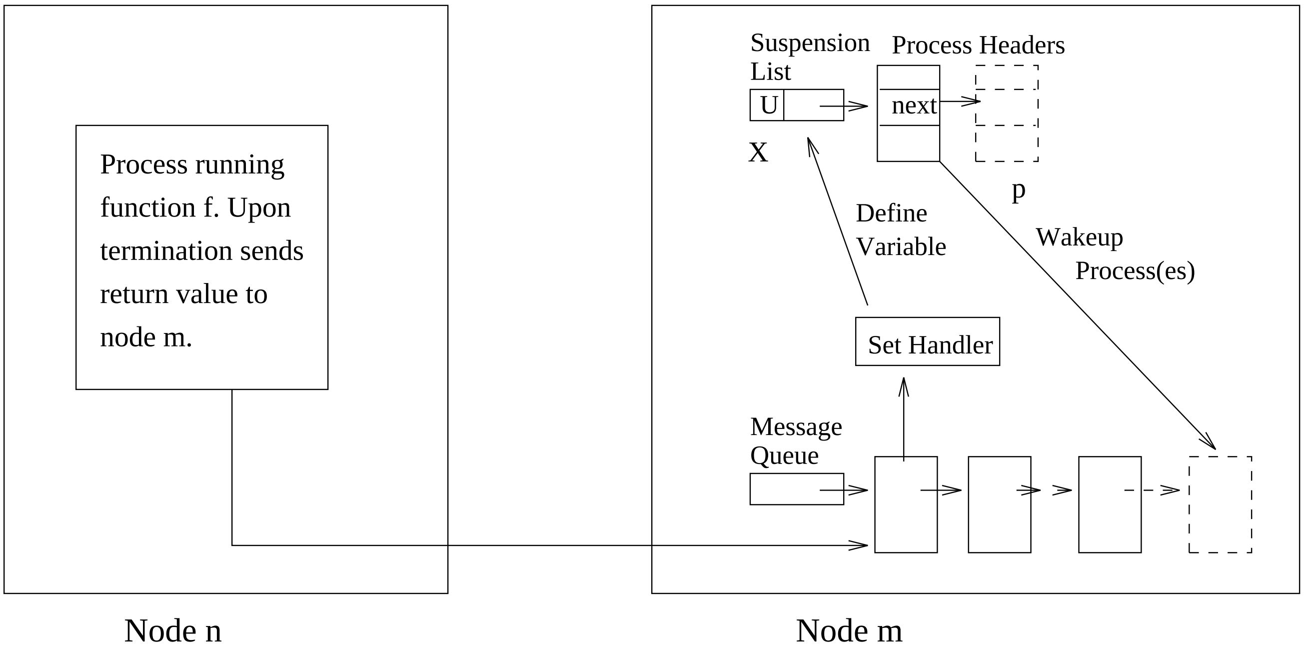

Figure 4.6: Waking Up a Suspended Process

Processes are created upon receipt of messages. Process creation consists of allocating a process header data structure. This header includes storage for the data portion of the incoming message, space to save the instruction pointer and registers, and a *next* field. The *next* field can be used to link a process header into a list. Recall that process suspension occurs when a process p executes $x = f(...)@n$, and attempts to access x before f has terminated. While process p is waiting for the variable x to be defined, x is tagged *undefined*, and contains the address of the first process header on a list of suspended processes. Each process header on the list contains the address of the next process header on the list in its *next* field.

Figure 4.6 shows a snapshot of the process suspension and awakening mechanism in operation. When the process p attempts to read the undefined variable x, a hardware fault is taken and a handler is invoked. This handler saves the instruction pointer and registers into the process header, adds the process header to the list of processes suspended on the variable x, and issues a hardware suspend instruction. Process p is now suspended waiting for x to be defined by a remote process executing function f. When the remote function terminates, it sends a message to the *set* handler on the computer containing p.

The *set* handler will define x to be the value returned by f, and wake up the processes suspended waiting for the value of x, including process p. The processes are woken up by sending the address of their process header to the *wakeup* handler.

4.6 Code Distribution

Recall that many of the applications being developed by the *Scalable Concurrent Programming Laboratory* require more than 1/4 megabyte of memory for code alone. The program code cannot be replicated at each J-Machine computer, and a mechanism for distributing a program's code across the machine is provided. The intent of code distribution is to maximize the memory available to user code at each computer. Recall that this mechanism utilizes bin-packing within the linker to minimize the amount of code that is resident at each computer.

Information regarding which distributed functions are present at a given computer is maintained using the hardware *associative memory*. A 32-bit function identifier *translates* into a pointer to a hash bucket containing a list of function header blocks. Each of these blocks contains the unique function identifier and the address of the beginning of the code. When code is requested from another computer, a function header is created, and the code address field is tagged *undefined*. Any attempts to locate the code prior to its arrival will cause the accessing process to suspend. Once the code is present, this field is updated with the starting address of the code, and subsequent references will return this entry point. The *receive* handler is responsible for waking up all processes suspended waiting for the received function. Code is purged from the cache when memory is exhausted. When code is being purged, the next function to be removed is that which was least-recently used.

Chapter 5

Evaluation

5.1 Performance Results

In terms of performance, the goals for this project can be stated as delivering hardware message-passing performance to application programs; providing an inexpensive code distribution mechanism; and generating high quality code.

The numbers for the hardware performance are derived from the results reported by the MIT Concurrent VLSI Architecture Project in [21]. The results reported here are based on a simple producer-consumer code. This code spawns a producer process on one computer which sends 100,000 messages, each of which creates a consumer process on another computer. This program appears in two forms: one-way communication as shown in Figure 5.1 and two-way communication as shown in Figure 5.2. The one-way version of the code creates 100,000 copies of the consumer, each of which increments a counter. When this counter reaches 100,000 the program terminates. In the two-way version of the code, each of the 100,000 consumers returns a value to the producer. When the producer has received all of the return values the program terminates.

Each program was compiled in three basic forms, based on the length of the messages generated. Message lengths of 8, 16 and 64 words were used. The distance that messages traveled varied based on the value of the constant *TARGET*. Messages were sent 1, 4, 8 and 14 hops. In addition, the two-way communication program was linked both with and without code distribution. Finally, a variation on each of the basic programs was created to approximate the overhead cost involved with executing the programs on one computer with no communication. This overhead represents the cost of creating processes, and executing the code. All of the resulting programs were executed 5 times each on the J-Machine using a 28MHz clock. After the programs had been executed, the 5 runs were averaged, and the cost of the overhead was subtracted from each of these averages. The resulting number was divided by 100,000 to provide the cost of communication per iteration. This number is reported, along with the actual hardware cost of sending a message of the appropriate length the correct distance.

```
#include "spawn.h"

void producer(void);
void consumer(int len, int *array);          /* No return value */

#define ITERATIONS 1000000                   /* Number of messages to send */
#define TARGET 8                             /* Computer to send messages to */

main()
{   producer()@0; }                          /* Start the producer on computer 0 */

void producer(void)
{   int i;
    int array[8] = {0, 1, 2, 3, 4, 5, 6, 7}; /* Data to Send */

    for(i = 0; i < ITERATIONS ; i++)         /* Send ITERATION messages */
        consumer(sizeof(array), array)@TARGET;
}

int counter = 0;                             /* Count the number of messages received */

void consumer(int len, int *array)
{   int i, j = 0;

    /* Message data can be used here without copying */

    if(++counter == ITERATIONS) exit(0);     /* Terminate program */
    else msgFree(&len);                      /* Release memory for this message */
}
```

Figure 5.1: One-Way Producer-Consumer

```
#include "spawn.h"

void producer(void);
int consumer(int len, int *array);              /* Returns an integer */

#define RUNS 2500                               /* Number of sets to execute */
#define ITERATIONS 40                           /* Number of messages to send per set */
#define TARGET 8                                /* Computer to send messages to */

main()
{   producer()@0; }                             /* Start the producer on computer 0 */

void producer(void)
{   int i, j, k;
    int array[8] = {0, 1, 2, 3, 4, 5, 6, 7};    /* Data to Send */
    int returns[ITERATIONS];                    /* Array for return values from consumer */

    for(k = 0; k < RUNS ; k++) {                /* Execute RUNS sets */
        for(i = 0; i < ITERATIONS ; i++)        /* Send ITERATION messages, expect a return */
                returns[i] = consumer(sizeof(array), array)@TARGET;

        for(i = 0; i < ITERATIONS ; i++)
                j = returns[i];                 /* Make sure all of the consumers are done */
    }                                           /* End of loop for RUNS */
    exit(0);
}

int consumer(int len, int *array)
{   int i, j = 0;

    /* Message data can be used here without copying */
    msgFree(&len);                              /* Release memory for this message */
    return j;                                   /* Return computed value */
}
```

Figure 5.2: Two-Way Producer-Consumer

5.1.1 One-Way Producer-Consumer Evaluation

Table 1 shows the data from executing the one-way producer-consumer code. The cost of sending a message from within a C program is only twice the hardware cost of sending the message. Compared to the cost of process creation and code execution, this communication cost is negligible. This is evident in the near-identical costs of sending messages any number of hops. This occurs because the largest element in the cost is executing the code, and the next message to be executed can be delivered before the target processor is ready to execute it. This data suggests that the messages spend significantly more time waiting to be executed in the message queue at the consuming processor than in transit from the producer.

	Hops	1	4	8	14
8 Word Messages	Hardware	4.53E-6	5.81E-6	6.09E-6	6.58E-6
	Software	1.50E-5	1.51E-5	1.53E-5	1.49E-5
16 Word Messages	Hardware	1.02E-5	1.04E-5	1.07E-5	1.11E-5
	Software	2.23E-5	2.24E-5	2.23E-5	2.24E-5
64 Word Messages	Hardware	3.79E-5	3.82E-5	3.85E-5	3.89E-5
	Software	7.94E-5	7.94E-5	7.94E-5	7.94E-5

Table 5.1: Timings for One-Way Communication, Average Latency in Seconds

5.1.2 Two-Way Producer-Consumer Evaluation

Without Code Distribution. Table 2 presents the timings for two-way communication, and reveals the cost of synchronization. This data indicates that performing synchronization has negligible cost. The increase in latency for larger messages is due to time spent waiting to deliver the outgoing messages. Once the incoming message buffer at the consuming computer has filled, the producer must wait until this buffer begins to empty to be able to complete a message send. The larger the outgoing messages, the longer it takes the producer to send them. The time difference between the two programs for 8 word messages is approximately twice the hardware cost for sending the return value message from the consumer back to the producer. These performance numbers indicate that our process suspension and awakening mechanism is efficient. As with the one-way communication, the cost of splitting the program up across computers is trivial in comparison to the cost of executing the code.

With Code distribution. Table 3 shows the performance differences for the two-way communication program with and without code distribution. Compared to code executed without using the code distribution mechanism the running time for 8 word messages increases by more than an order of magnitude; 16-word messages perform slightly better; and 64 word messages show a slowdown of only 1.5. The improvement in relative times for 64 word messages is due to the cost of code distribution being dwarfed by

	Hops	1	4	8	14
8 Word Messages	One Way	1.50E-5	1.51E-5	1.53E-5	1.49E-5
	Two Way	2.30E-5	2.30E-5	2.30E-5	2.30E-5
16 Word Messages	One Way	2.23E-5	2.24E-5	2.23E-5	2.24E-5
	Two Way	3.42E-5	3.42E-5	3.41E-5	3.39E-5
64 Word Messages	One Way	7.94E-5	7.94E-5	7.94E-5	7.94E-5
	Two Way	3.20E-4	3.20E-4	3.21E-4	3.21E-4

Table 5.2: Timings for One-way versus Two-Way Communication, Average Latency in Seconds

the amount of time this program spends waiting to complete message sends. This simple producer-consumer program does not provide any opportunity for the system to mask the latency of code fetching by overlapping communication and computation. These numbers demonstrate that a poor usage of the code distribution mechanism can be very expensive. In this example, the functions **consumer** and **msgFree** should be replicated functions as they are each called 100,000 times.

An examination of the code distribution mechanism suggests an optimization based on reducing the cost of locating a replicated function that is already present at the current computer. For every call to a distributed function, the code distribution mechanism generates a call to a C routine to locate the function in the *associative memory cache*. The overhead of calling this routine, primarily creating and destroying a call frame data structure, is high. The efficiency of code distribution will be improved by inserting the cache check directly into the kernel. This optimization will decrease the cost of code distribution by approximately 80%. This optimization is not yet implemented.

	Hops	1	4	8	14
8 Word Messages	With	3.96E-4	3.96E-4	3.96E-4	3.96E-4
	Without	2.30E-5	2.30E-5	2.30E-5	2.30E-5
16 Word Messages	With	4.11E-4	4.11E-4	4.11E-4	4.11E-4
	Without	3.42E-5	3.42E-5	3.41E-5	3.39E-5
64 Word Messages	With	4.85E-4	4.85E-4	4.88E-4	4.88E-4
	Without	3.20E-4	3.20E-4	3.21E-4	3.21E-4

Table 5.3: Timings for Two-Way Communication with and without Code Distribution, Average Latency in Seconds

5.2 Application Example

This section describes an example piece of application code that has been run on the J-Machine. This program has all of the interesting characteristics of applications being developed by the *Scalable Concurrent Programming Laboratory*, and in particular utilizes a standard set of libraries which provide generic support for organizing applications. This code uses domain decomposition to solve Laplace's equation $\nabla^2\theta = 0$ using sinusoidal boundary conditions. Each process within the program communicates with other processors at computers in close proximity. Barrier synchronization is performed at every timestep.

```
FORALL computers n {
   local norm = initialize();
   global norm = barrier(localnorm);
   termination condition = EPSILON * global norm;
   FORALL time t {
      while(global norm > termination condition) {
           Send faces to all neighbors;
           Receive faces from all neighbors;
           local norm = timestep(faces);
           global norm = barrier(local norm);
      }
   }
}
```

Figure 5.3: Parallel Dirichlet Algorithm

The algorithm being executed is summarized in Figure 5.3. All computers are initialized, and a barrier is used to ensure that all computers have completed initialization before continuing. A termination condition is calculated, and processing begins. Until the termination condition is satisfied, a series of timesteps are executed. At each timestep, a face in each computer is sent to its neighbors. After a computer has received messages from all of its neighbors, the local norm is computed. Another barrier is used to recompute the global norm across the whole machine. The structure of this program is illustrated in 5.4, which is based on discussion in [7]. The domain of the problem is broken up into partitions. Each partition is mapped onto a computer. The exchange of information for each timestep requires the transmission of the edges adjacent to a partition boundary being sent to the computer containing the neighboring partition.

Using the techniques described in this thesis, this algorithm is implemented as a set

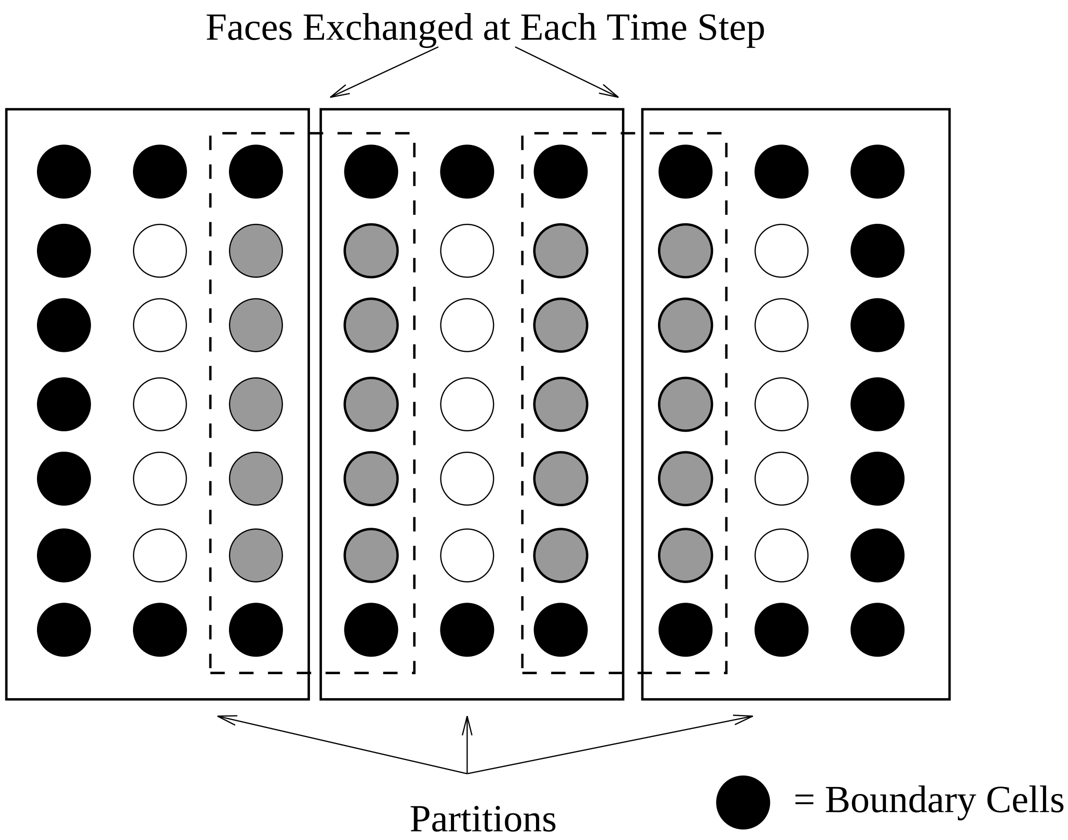

Figure 5.4: Abstract Picture of the Dirichlet Algorithm

of handlers. These handlers are shown in Figure 5.5.

Initialization. The *startnodes* handler runs at every computer, and for each virtual node at a given computer reads in an initialization file, initializes local state, and then calls a library routine which implements a barrier. The **barrier** routine maintains a counter, and ensures that all virtual nodes have finished executing *initial_setup*, which calculates the termination condition and initializes the count of expected messages, prior to invoking *timestep_send* at all nodes.

Communication. At the beginning of each timestep, the *timestep_send* handler is invoked at all virtual nodes. This handler works through a list of *port* pairs, consisting of a *sendport* representing an outgoing port for the current node, and a *recvport* which is the target computer for the *sendport*. For each of these pairs, a msg is assembled containing the *edge* that needs to be sent to the *sendport*. This message is used as the argument to the *timestep_recv* handler, which is invoked at computer *recvport* using the mapping annotation @. Each time that *timestep_recv* is invoked, it saves a pointer to the incoming message, and decrements a counter of expected messages. When there are no more messages pending at a particular computer, the code to execute a timestep is run. When the timestep is complete, a second barrier is called to check for termination, and then initiate the next timestep by calling *timestep_send*.

Termination At the conclusion of each timestep, the *terminate* handler is invoked. This handler compares the current global norm against the termination condition. If this comparison indicates that the computation has terminated, the program exits. Otherwise, the count of expected messages is reset.

```
startnodes(...) {
    while(nodes for this computer) {
        read graph description file
        localnorm = initialize(node)
        barrier(localnorm,initial_setup,timestep_send)
    }
}

initial_setup(node,globalnorm) {
    term_cond = EPSILON * globalnorm
    pending = RECVPORTS
}

timestep_send(node) {
    foreach pair (sendport,recvport) {
        msg=get_face(sendport)
        timestep_recv(msg)@recvport
    }
}

timestep_recv(node,msg) {
    nodemsgs = msg
    decrement pending
    if(no messages pending) {
        localnorm=timestep(...)
        barrier(localnorm,terminate,timestep_send)
    }
}

terminate(node,globalnorm) {
    if(globalnorm > term_cond)
        pending = RECVPORTS
    else
        exit()
}
```

Figure 5.5: Dirichlet Example Message Handlers

Chapter 6

Experience with the Message-Driven C System

Recall that the original goals of this project were to allow for message sending and receiving, as well as process suspension and awakening, without having to perform copying. This would allow hardware performance to be delivered to the application codes. In general, both the J-Machine hardware and the programming system we have been able to build are quite satisfactory. However, there were a number of idiosyncracies in the hardware which complicated our task. These concerns are being addressed in the M-Machine, the next generation machine currently being designed by the MIT Concurrent VLSI Architecture Group. This section describes and discusses what was learned about the J-Machine in the course of this experiment. This section also contains a discussion of some unforeseen problems with the compiler which might be of interest to others wishing to retarget GCC for alternative systems.

6.1 Remote Function Invocation

Conceptually, in the *Message-Driven C* computational model, function arguments reside only within a received message, and no copying of the data is required upon message reception. It was possible to construct a system that achieved this functionality, however this necessitated considerable contortions. This mechanism became unmanageable when combined with supporting process suspension, and there was no choice but to accept copying on receive. It is not possible to leave arguments in the message buffer because the J-machine hardware distinguishes between message buffer memory, and general memory. The portion of memory used for message reception is treated as a circular queue. There are 3 special properties of this piece of memory:

- When accessed through register A3, the hardware handles addressing of messages that wrap around past the end of the buffer.

- When accessed via register A3, a hardware fault is generated if an attempt is made to read a data word that has not yet been received.
- When the process created in response to a message reception suspends, the storage for that message is reused by the hardware.

The hardware wraparound issue was dealt with by defining a special data type, a *wrapped pointer*, which is a pointer to a data item that wraps around from the end of the queue to the front. Any access to a *wrapped pointer* results in the execution of a fault handler which analyzes the instruction causing the fault and modifies and/or reissues this instruction as necessary.

The issue of accessing data that has not yet been received is not a problem if message data is accessed via A3, as this allows the hardware fault mechanism to properly signal attempts to access unreceived data. The method of dealing with this type of fault is to back up the instruction pointer and reissue the faulting instruction until it no longer faults: this indicates that the data has been correctly received.

Although the data in a message will be available in the message buffer when the process is initiated, once the process suspends the hardware can overwrite the portion of the message buffer containing this data. To support suspension it is necessary to guarantee that a process can find its arguments both before and after it has suspended. This requires copying the message data upon either message reception or process suspension. Although a mechanism to allow a process to access its arguments properly prior to suspension was developed, this is not useful if it is not possible to also guarantee that the process will still be able to access its data after suspension. In order to make this guarantee, all pointers into the message buffer would have to be updated to reflect the change in location of the data following the copy. As the cost of locating all such pointers is prohibitive due to the loose constraints on pointer copying in C, it is necessary to perform a *copy on receive*.

A preferred approach would be for the hardware not to distinguish between general memory and message buffers. There would consequently be no difficulty with allowing received messages to remain where they were originally placed upon reception, and it would not be necessary to copy on either receive or suspend.

6.2 Compiler Problems

Aside from the usual problems attendant to deciphering any large piece of software, the difficulties that encountered with GCC were all related to assumptions about the target architecture which the compiler makes. Even though GCC is designed to be retargetted for a wide variety of machines, there are certain architectural features which can make this process very difficult. These items could be better addressed in a compiler written specifically for fine-grain multicomputers.

Registers. The greatest difficulty associated with GCC was that it expects more general registers than the 3 available at each Message-Driven Processor. This was manifest in many ways: excessive numbers of local variables are created for temporary registers; function values must be returned indirectly because returns in registers would occasionally be overwritten; double-precision argument passing was problematic because each double-precision argument requires two data registers and this does not leave enough registers to do data manipulation.

Stack-Based Architecture. Although GCC does provide enough support to allow the use of heap-based frames rather than a stack, it does not do so without cost. The main component of this cost is the necessity to do special-case handling for nested calls to library functions. This occurs because outgoing function arguments are placed into specific slots, rather than placed onto a stack. For example, a process could place arguments into the first three arguments slots, and then discover that it needs to call a library routine. The arguments to the library routine (e.g. double-precision arithmetic) could require the use of the first four argument slots. It is therefore necessary to preserve the contents of the first three argument slots prior to the library call, and restore them afterwards.

Position Independent Code and Data. GCC handles references to functions as if they are constants. This is a poor assumption within this system: code can be transported from one computer to another and placed on the heap in a location that is not known until runtime. Although the compiler does have support for Position Independent Code, to support systems such as dynamic linking on the Sparc, this requires an address register. The J-Machine does not have enough address registers for one to be available for this purpose. A set of linker techniques were developed to work around this problem. These techniques assign identifiers to functions and patch all references to functions with their identifiers. Similarly, the linker must calculate the addresses of global variables, and ensure that these are also correct within the code. Unfortunately, without expanding the microkernel to allow for intervention every time a global variable is accessed it is necessary to permanently allocate all global and static data in the same place at every computer. This makes it impossible to follow the more desirable route of transporting static data for a function with the code for the function.

Lack of ANSI Compliance. Surprisingly, when using standard ANSI compliant test suites, such as Metaware, it was discovered that GCC is not an ANSI compliant compiler. This complicated the running of validation suites, as well as hindering the evaluation of validation failures.

6.3 Code Generation

A variety of problems occurred in code-generation that were specifically linked to the design of the Message-Driven Processor.

Integer Division. Recall that the computational model provides an abstraction of the J-Machine as a collection of consecutively numbered computers. To support this abstraction, it is necessary to be able to translate from an integer computer number to a physical address. This operation requires several integer divisions and integer modulo operations. The cost of software division and modulo is high, making this conversion expensive. While it is possible to work around this problem by having a conversion table for virtual to physical computer addresses, such a solution will obviously not scale, and is expensive in memory usage.

Relative Addressing The instruction set does not allow the use of a negative constant offset for relative addressing. Rather than simply using:

```
move r2, [-1, a2];
```

It is necessary to use an additional instruction and register:

```
move -1, r0;
move r2, [r0, a2];
```

Address Data Type and Address Registers. As this machine is intended to function under an *object-oriented paradigm,* the address data type contains both a beginning absolute address for a data item, and a length of that item. If the length is non-zero, hardware bounds-checking is performed on all references through the pointer. The physical layout of these pointers has the length as the low-order 10 bytes, with the address above it. One result of this structure is that adding an integer to a pointer does not produce the desired result. In addition, the Message-Driven Processor distinguishes between data registers and address registers. Relative addressing can only be done using an address register as the base of the address. Moves to and from memory must have a data register as the non-memory operand. Most operations require that at least one of the operands be a data register. The contents of an address register must be an address data type to be useful. Address data types are extremely limited in their usefulness when they are in data registers. One example of the code generation problems caused by these issues can be seen when we want to add 4 to the address contained in register A1. We cannot do:

```
add a1, 4, a1
```

But must instead use 7 instructions and a scratch register:

```
move a1, r0          ; Move address register A1 into register R0
wtag r0, INT, r0     ; Tag contents of R0 to INTeger
ash r0, -10, r0      ; Shift out size field from address
add r0, 4, r0        ; Add 4 to contents of R0
ash r0, 10, r0       ; Shift 0 into size field of address
wtag r0, ADDR, r0    ; Tag contents of R0 to ADDRess
move r0, a1          ; Move register R0 into address register A1
```

Byte Addressing. The Message Driven Processor is a 32-bit processor, with memory addressable in 32-bit words. There is no support provided for efficiently accessing the individual bytes of a word. One of the ramifications of this is that strings must be either inefficient in time due to requiring special handling to treat one 32-bit word as 4 8-bit bytes; or they must be inefficient in space by using one 32-bit word for each character in the string. The solution to this problem used in this system is to have the compiler generate code that is inefficient in space, but provide a packed-string library to support strings that are inefficient in time.

Processor Status Flags. The processors have a collection of status flags that control such features as interrupts enable/disable, checked/unchecked mode, faults enable/disable and whether or not register A3 points into the message queue. These flags can only take values of 0 or 1. It is not possible to move a value directly into one of these flags, one can only move the contents of a data register into a flag. Instead of doing:

```
move 1, U
```

One must use two instructions and a scratch register:

```
move 1, r0
move r0, U
```

6.3.1 Floating Point

The Message-Driven Processor does not provide any hardware support for floating-point computations. Although there are aspects of the architecture that are geared towards alleviating this problem, the emphasis in these features is on fast single-precision computations. As most of the applications being worked on within the SCP group use IEEE double-precision representation, it was necessary to provide software emulation of this standard. In the process of creating the support for this emulation, a number of problems were encountered.

The most striking of these problems deals with the inadequacy of the MDP register file. This lack of registers is exacerbated by faults in the register allocator of GCC. The most prevalent problem deals with the issue of returning a double-precision value from a function. Ideally, return values will be passed back to the caller in registers. In the case of a double-precision number, this requires two registers. As previously mentioned, there are only three registers available to the allocator. As a result of this, as soon as the function value is returned, it must be immediately moved into memory. This process of moving the value is complicated because under many circumstances GCC generates code that overwrites one of the return words with an offset into memory. This problem required special code to be added to the compiler so that under certain circumstances the decisions of the register allocator are over-ruled.

As an example, the compiler might try to generate the following piece of code (note that **DFDIV** performs double-precision division, and returns an answer in R2, R3):

```
move -11, r0        ; Generate offset for local variable
move [r0, a1], r3   ; Fetch address for storage of return value
CALL DFDIV          ; Call the division routine
move r2, [r3,a0]    ; Store the first word of the result
add r3, 1, r3       ; Increment the address
move r3, [r3,a0]    ; Store the second word of the result
```

There are two major problems with this code. First of all, the return value for the division routine overwrites the target address into which it should be stored. Secondly, an increment operation is performed on one of the words of the return value. To correct these problems, it is necessary to preserve the value of R3 across the division call, and modify what register is used for the store operations. These modifications result in the following (less efficient) code:

```
move -11, r0        ; Generate offset for local variable
move [r0, a1], r3   ; Fetch address for storage of return value
move r3, id3        ; Store address value in register ID3
CALL DFDIV          ; Call the division routine
move id3, r1        ; Place the address value into R1
move r2, [r1,a0]    ; Store the first word of the result
add r1, 1, r1       ; Increment the address
move r3, [r1,a0]    ; Store the second word of the result
move r1, r3         ; Place the address value back into R3
```

While this code will perform the correct task, it requires 3 additional instructions, and 1 additional register.

6.4 High-Level Language Design

Part of this research project involved investigating the construction of a high-level language system on top of *Message-Driven C*. This work involved designing a set of modifications to the PCN Runtime System [13] to take advantage of the hardware capabilities of the J-Machine. In addition, research was done into extensions to the PCN source language that would allow optimizations to take greater advantage of the hardware. By knowing the form of the generic programming techniques, it is possible to construct new *communication-oriented compiler optimizations* that capitalize on the structure of the technique in use. These optimizations take advantage of the hardware without changing the basic programming semantic. For example, in Figure 6.1 additional information in the form of an abstract data type *stream* has been provided. This signifies to the compiler that a particular stream protocol is in use. Furthermore, the direction of communication, which is from the producer to the consumer, can be determined from the function prototypes. This information makes it possible to generate code that allows the processes to communicate without an explicit representation of the list structure, thereby avoiding all overheads associated with structure copying. The list notation only signifies the constraint that message order is preserved.

6.4.1 Example High Level Program

Figure 6.1 shows a generic producer-consumer program that illustrates many of the programming concepts used in the construction of concurrent algorithms. Execution begins in computer 0 at the main function. Initially, a stream is created to carry messages from the producer to the consumer (1) and the global array A is initialized (2). Subsequently, two concurrent processes are spawned: The first is a producer that executes at computer 0 (3); The other is a consumer that executes at computer 1 by virtue of the mapping annotation @ (4). Notice that the producer and consumer share the communication stream S. Only when both processes have terminated, does the tidyup function execute (5). Upon its termination the entire program terminates.

The producer sends a single message to the consumer containing a copy of the array A (6). It then recursively sends n−1 further copies of the array (7) until termination occurs and the stream is closed (8). The consumer *suspends* until a message is received by virtue of the matching operation (9). It subsequently uses the message (10), and finally consumes any remaining messages recursively (11). Eventually, the consumer terminates when the stream is closed.

Figure 6.2 outlines how this protocol could be implemented using the J-machine tagging structure and a collection of message handlers provided by MDC. The handlers are trivial to implement using the systems programming layer described in Section 4.3. In this figure, the producer is signified by the process P and the consumer by the process C.

The concurrent processes are compiled into two calls to the spawn function. The first is spawned at computer 0 (the current computer) and simply invokes the producer; the second causes the consumer to be spawned at computer 1. In order to establish the shared stream S, the producer is suspended until the location of S at the consumer in computer 1 becomes *known*. Similarly, the consumer suspends until the first message produced by the producer is *known* (A).

Eventually, the stream location becomes *known* by virtue of communication sent from computer 1 and the producer is then scheduled (B). The producer now begins to send messages to the consumer. The first message causes the consumer to be scheduled (C). If further messages arrive before that consumer is executed they are simply associated with the stream variable (D). Eventually, the consumer becomes the current process by reaching the front of the message queue. At this time it may, using a tail recursion optimization, iteratively consume all of the available messages and suspend to await further messages.

```
int A[50];

main()
{   stream S;                        /* 1 */
    initialize(A);                   /* 2 */
    {|| producer(10,S);              /* 3 */
        consumer(S)@1;               /* 4 */
    }
    tidyup();                        /* 5 */
}

producer(int n, out stream S)
{   if(n > 0) {
        S = [A | Ss];                /* 6 */
        producer(n-1,Ss);            /* 7 */
    }
    else
        S = [];                      /* 8 */
}

consumer(in stream S)
{   if(S ?= [A|Ss]) {                /* 9 */
        use(A);                      /* 10 */
        consumer(Ss);                /* 11 */
    }
}
```

Figure 6.1: Generic Producer-Consumer Protocol

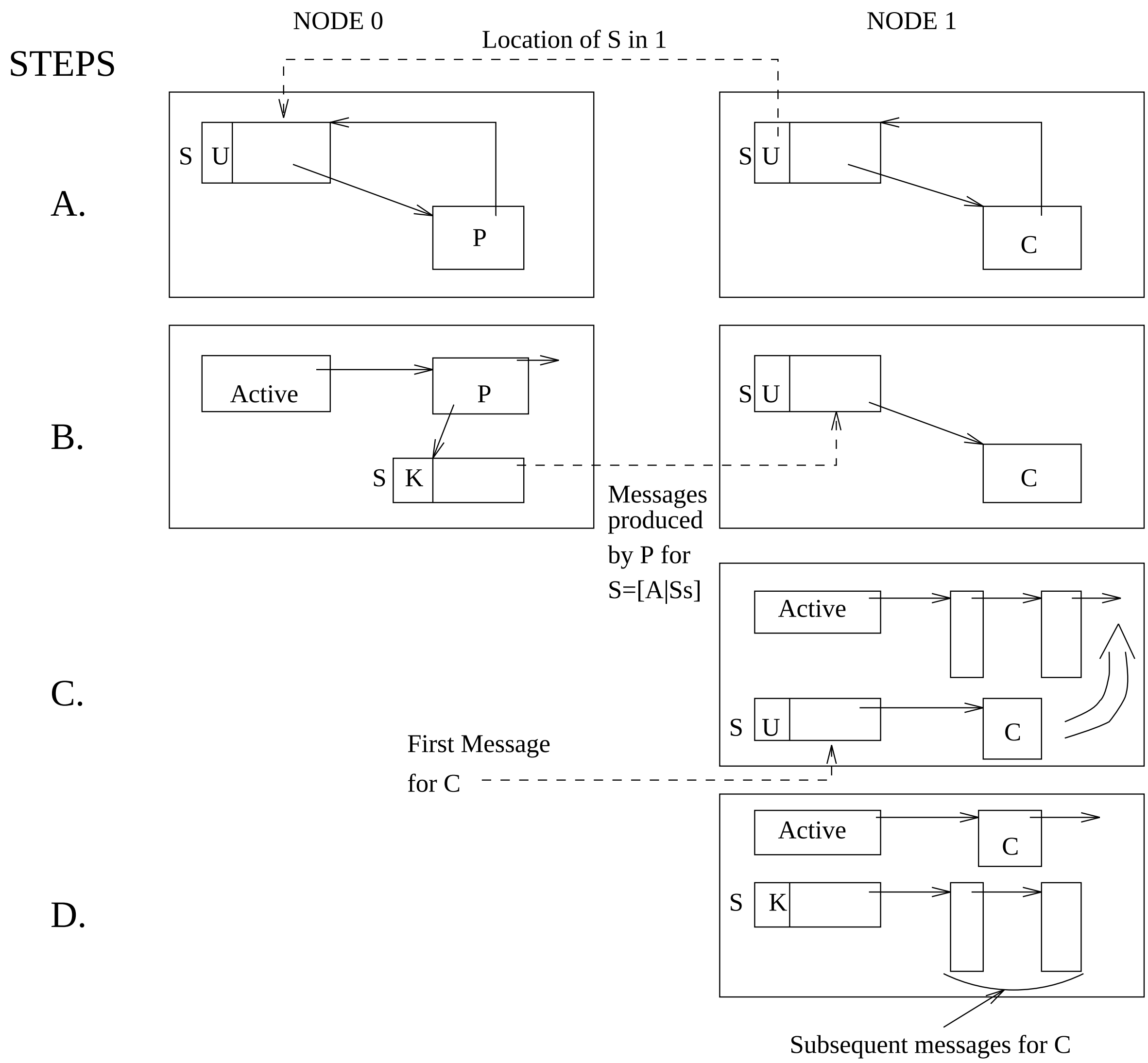

Figure 6.2: Stream Communication

A simple counting mechanism is used to provide a barrier that detects termination of the two concurrent processes. A counter is associated with entry to a parallel block, this counter is initialized to the number of processes spawned. Both the producer and consumer are provided with the location of this counter. When a process terminates it uses communication to decrement the counter. Eventually, termination is signified by a counter value of zero.

All of the operations we have described involve only one way communication. Further, all of the synchronization operations can be built on top of variables that are in some binary state *known* or *unknown*. This form of synchronization is precisely that provided by the tagging structure of the J-machine.

The producer-consumer protocol is the simplest protocol to implement. However, minor generalizations of the basic implementation strategy described here allow all of the other basic stream-based programming techniques to be implemented. These implementation strategies are the subject of ongoing research.

Chapter 7

Future Directions

As with any other research project, there are several interesting questions that arose in the course of this research that were not pursued. This section covers some of these problems.

7.1 Code Partitioning

The current implementation of the linker uses a simple bin-packing algorithm to place *distributed* functions on *home computers*. The question of what is the best way to perform this mapping is open. A possible line of research would be to provide the linker with the ability to receive dependence information from the compiler. This information could be used to minimize the distance that code needs to travel, as well as possibly placing functions on more than one computer if space is available. In addition, functions could be placed into groups so that code transport sends multiple functions in a situation where a request for one function guarantees that additional functions will soon be required. Also, it would be interesting to learn if there might not be other types of functions that the system should support to provide greater efficiency while executing a distributed program. For example, it might be desirable to have a function that is distributed and assigned to one home computer, but which becomes permanently resident at any computer to which it is transported.

7.2 Communication-Based Compiler Optimizations

Although this current work has advanced knowledge of how to compile and run programs on fine-grained multicomputers, it has not fully addressed the issue of how to efficiently do so. Future work could focus on developing a set of global, communication-based optimizations to improve the quality of the scheduling of an instruction stream. These optimizations could be implemented using semantics-preserving source-to-source transformations.

An example of such a communication-based optimization is a transformation of a barrier, such as the one shown in Figure 5.3, to decrease the serialization which generally occurs when performing this type of synchronization. A barrier code might typically look like:

```
int messages;

barrier()
{   Change global state in response to incoming message;
    if(messages - 1 = 0)
        continuation();
}

continuation()
{   Proceed with computation;
    Assume that global state is fully updated;
}
```

Figure 7.1: Sample Barrier Code

Each time a message to the barrier is received, a counter is decremented. When the counter reaches zero the continuation function is called at each computer. The naive way to compile this functionality would be to treat the barrier and the continuation as two distinct routines, and have the barrier code perform a sequential call to the continuation function when the counter reaches zero. If an analysis is performed on the data dependence between the barrier and continuation code, it would be possible to detect situations in which the continuation code can begin execution prior to the end of the barrier code. In the Message-Driven C system there are high overhead costs associated with function calls. Being able to execute the code necessary for performing the function call for the continuation function in parallel with execution of the barrier code would yield an improvement in program execution. This occurs because the processes executing the continuation code would normally be idle until barrier had completed. The transformed code would execute the two routines shown in Figure 7.2 in parallel.

```
int messages; /* Count of messages expected */
int syncFlag; /* Flag to coordinate between barrier and continuation */

barrier()
{   Change global state in response to incoming message;
    if(messages - 1 = 0)
                set syncFlag;
}

continuation()
{   Perform initialization that does not depend on barrier;
    Wait for syncFlag to be set;
    Proceed with computation;
    Assume that global state is fully updated;
}
```

Figure 7.2: Transformed Barrier Code

Chapter 8

Conclusion

This work demonstrates that the specialized hardware in a fine-grained multicomputer such as the J-Machine can be effectively used to implement communication and synchronization in a concurrent program without incurring large software overhead costs. The direction of research currently being pursued by the Concurrent VLSI Architecture Group at MIT, when coupled with the software techniques being developed by the Scalable Concurrent Programming Laboratory at Caltech, gives hope that it may be possible to construct a parallel-programming system in which processor utilization is high without significant execution time being devoted to software overheads. The techniques described in this thesis are an important step in this direction. The major contributions of this project have been the development of compiler and runtime techniques that effectively use existing hardware mechanisms; and the evaluation of the existing hardware to assist in the design of the next generation of fine-grained machines.

The implementation techniques allow new architectural features to be accessed directly via native code compilation. Hardware performance is delivered directly to applications by removing software overheads associated with message-passing. Messages are copied directly into the network on sending, and processes may execute directly out of message buffers on receiving. Code and data may be distributed through a combination of linkage and run-time microkernel support. Communication latency is hidden by a process suspension mechanism. Processes utilize a heap-based allocation scheme rather than a stack and thus may suspend without copying. Although some aspects of these concepts have been found awkward to implement on the J-Machine, these issues have been resolved in designs currently under construction at MIT.

The main concepts described in this thesis have been implemented and are in use on prototype J-machines at Argonne National Laboratory, Caltech, and MIT. This system is currently being used for large scale application development by more than 30 scientists at Argonne National Laboratory, Caltech, MIT and The Aerospace Corporation. The model here advocated for concurrent programming is not new and has been used extensively for applications development in a variety of systems developed by the *Scalable Concurrent Programming Laboratory* and others. The contribution of this work rests on new and simple implementation techniques for fine-grain architectures.

Bibliography

[1] Agha, G., *Actors*, MIT Press, 1986.

[2] Agha, G., et. al., *Abstraction and Modularity Mechanisms for Concurrent Computing*, IEEE Parallel and Distributed Technology, May, 1993.

[3] Arvind and Nikhil, R.S. "Executing a Program on the MIT Tagged-Token Dataflow Architecture," Lecture Notes In Computer Science 259, 1987.

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

[5] Boden, Nanette J., *Runtime Systems for Fine-Grain Multicomputers*, Ph.D. dissertation, California Institute of Technology, Department of Computer Science, 1993.

[6] Canetti, R., et. al., "The parallel C (pC) programming language, " *IBM Journal of Research and Development*, 35(5/6):727-741, September/November, 1991.

[7] Chandy, K. M., and Taylor, S., *An Introduction to Parallel Programming*, Jones and Bartlett, 1991.

[8] Chien, Andrew, "Supporting Modularity in Highly-Parallel Programs", in *research Directions in Object-Based Concurrent Systems,* MIT Press, 1993.

[9] Chien, A., Karamcheti, V., and Plevyak, J., "The concert system - compiler and runtime support for efficient fine-grained concurrent object-oriented programs," Technical Report, UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[10] Dally, W. J., et al., "The J-Machine: A Fine-grain Concurrent Computer," *Information Processing 89*, G. X. Ritter (ed.), Elsevier Science Publishers B.V., North Holland, IFIP, 1989.

[11] Darnell, Peter A., Margolis, Philip E., and Taylor, Stephen, *Software Engineering in C*, Springer-Verlag, revised edition in progress.

[12] Foster, I. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1989.

[13] Foster, I., and Taylor, S., "A Portable Run-Time System for PCN," in Argonne National Laboratory Technical Memorandum No. 137, ANL/MCS-TM-137, January, 1990.

[14] Foster, I., and Taylor, S., "A Compiler Approach to Scalable Concurrent Program Design", ACM Transactions on Programming Langauges and Systems (to appear).

[15] Gehani, N.H., and Roome, W.D., "Implementing Concurrent C, " *Software Practice and Experience*, 22(3):265-285, March, 1992.

[16] Horwat, W., *Concurrent Smalltalk on the Message-Driven Processor*, Masters thesis, Massachssetts Institute of Technology, Computer Science Department, September, 1991.

[17] Horwat, W., *Message-Driven Processor Simulator*, MIT Concurrent VLSI Architecture Memo 38, Massachssetts Institute of Technology, Computer Science Department, May, 1991.

[18] Marlin, C.D., "A Heap-Based Implementation of the Programing Language Pascal," *Software Practice and Experience*, 9(2):101-119, February, 1979.

[19] Maskit, D., Taylor, S., *Experiences in Programming the J-Machine*, California Institute of Technology, Department of Computer Science Technical Report, CS-TR-93-11, 1993.

[20] Maskit, D., et. al., *System Tools for the J-Machine*, California Institute of Technology, Department of Computer Science Technical Report, CS-TR-93-12, 1993.

[21] Noakes, M., Wallach, D. and Dally, W., "The J-Machine Multicomputer: An Architectural Evaluation," *Proceedings of the 20th ' International Symposium on Computer Architecture*, May, 1993.

[22] Schauser, K.E., Culler, D.E. and von Eicken, T., "Compiler-Controlled Multithreading for Lenient Parallel Languages," Lecture Notes In Computer Science 523, 1991.

[23] Seizovic, Jakov N., *The Architecture and Programming of a Fine-Grain Multicomputer*, Ph.D. dissertation, California Institute of Technology, Department of Computer Science, 1994.

[24] Seitz, C. L., "Multicomputers," *Developments in Concurrency and Communication*, C.A.R. Hoare (ed.), Addison-Wesley, 1990.

[25] Su, W., *Reactive-Process Programming and Distributed Discrete Event Simulation*, California Institute of Technology, Department of Computer Science Technical Report, CS-TR-89-11, 1990.

[26] Taylor, S., *Parallel Logic Programming Techniques*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[27] Thinking Machines Corporation, Cambridge, MA. *CM5 Technical Summary,* October, 1991.