Self-Distributing Computation

by

Thomas R. Woodfin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author .....
Department of Electrical Engineering and Computer Science
May 13, 2002

Certified by.....
Thomas F. Knight, Jr.
Senior Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Self-Distributing Computation

by

Thomas R. Woodfin

Submitted to the Department of Electrical Engineering and Computer Science
on May 8, 2002, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

In this thesis, I propose a new model for distributing computational work in a parallel or distributed system. This model relies on exposing the topology and performance characteristics of the underlying architecture to the application. Responsibility for task distribution is divided between a run-time system, which determines when tasks should be distributed or consolidated, and the application, which specifies to the run-time system its first-choice distribution based on a representation of the current state of the underlying architecture. Discussing my experience in implementing this model as a Java-based simulator, I argue for the advantages of this approach as they relate to performance on changing architectures and ease of programming.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist

Acknowledgments

Thanks to Tom Knight and the rest of the Aries group for the inspiration for this thesis and everything one could want to know about novel architectures.

The CSED group at the Institute for Defense Analyses earned my great appreciation for their insightful comments and feedback on an early presentation of this work.

MIT wouldn't have been half as much fun without the antics of Aaron Adler, Andrew Montgomery, John Hernandez, Gary Mishuris, Sid Nargundkar, John Bartocci, Nickolai Zeldovich, and the rest of the D12 crew.

Thanks to my family for their eternal support, and my friends for keeping me going–yes Carla, I wrote my 10 pages today!

And a special thanks to Jen Eppig: the promise of her company inspired me to finish this as soon as possible.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

We introduce a new programming model for parallel and distributed systems, Self-Distributing Computation (SDC). In the SDC model, the responsibility for distributing and parallelizing an application is split between the programmer and a run-time system. The programmer is responsible for explicitly describing how a computation can be divided into parallel components and distributed, as well as how these components can be recombined. The run-time system is responsible for determining which computations should be distributed and recombined, as well as providing to the application an abstract model of the underlying architecture upon which to distribute itself.

The advantages of this approach include increased flexibility for the computer architect, and better abstraction, performance and portability for the programmer.

A Java implementation of an SDC model architecture, called Mimoid, is also described. The implementation provides a useful testbed for experimenting with this new model, and the lessons learned from programming and implementing this model are discussed.

Chapter 2

Motivation

There is a constant drive in many areas of research, industry, and government for more and more computing power. Each new generation of computers inspires new applications with more demanding requirements in a seemingly endless cycle.

2.1 High performance applications

Researchers in a variety of fields require massive computational power to perform and analyze ever more accurate simulations. In the biological context, IBM is constructing a massive computer to simulate the folding of proteins, calculating the evolution of atomic interactions on a quantum scale[11]. On the other end of the scientific spectrum (but still from IBM), the Department of Energy's ASCI-White computer simulates the relativistic effects of the massive release of energy from a nuclear explosion[2].

In industrial applications, the rapid growth of the Internet has led to a demand for web servers to handle greater capacity and databases capable of processing complex transactions more rapidly. The premiere web search facility, Google, receives over 150 million requests a day[1]. The predicted trend towards web services, program-to-program interactions over the web infrastructure, promises to increase the computational requirements of network requests. Generating, encoding and decoding multimedia content also involves computation-intensive algorithms and massive data

sets.

2.2 Increasing computational power: two tracks

2.2.1 Building faster processors

There are two general tracks to obtaining more computational power. Moore's "Law," an empirical observation that the number of transistors that can be integrated onto a single chip doubles roughly every 18 months, has held true since it was first proposed. With the increase in transistor count usually comes a comparable increase in performance, and while there have been perennial proclamations of the imminent demise of this trend, Intel and other semiconductor manufacturers continue to produce faster chips at or beyond this rate.

Unfortunately, the complexity involved in designing chips with billions of transistors is massive, both from management and manufacturing perspectives. Intel's latest chips require design and production teams of tens of thousands of engineers, as well as fabrication facilities with costs well over a billion dollars. The effort required to keep up with Moore's Law may remain technologically feasible, but it is unclear how long economic incentives will exist to justify the expenditure[4].

2.2.2 Connecting multiple processors

The other track to faster computers is harnessing the power of multiple processors, either connected directly in a single machine, or multiple independent machines connected by a network. These systems are known, respectively, as parallel and distributed computers. The appeal is obvious: although "state-of-the-art" processors are extremely expensive to design and manufacture, economies of scale make individual processors relatively inexpensive. It may have cost Intel several billion dollars to make the first Pentium IV, but marginally on the order of $100 to make the second.

The challenge, of course, is to efficiently use the available computing power. While it is theoretically possible to get twice the performance out of two processors, or a

thousand times the performance out of a thousand, or a million times out of a million, this level of efficiency is unattainable for most problems. However, the relative expense of building more chips vs. building faster ones is such that even a much lower efficiency would be a valuable result.

Much work has been done on the hardware side to develop efficient means of connecting multiple processors, both locally and over a network. A vast variety of interconnects and routing protocols have been developed, some suited to particular programming approaches more than others.

2.3 The software problem

The challenge is writing software to exploit these designs. There are two general methods for approaching the optimal solution for a parallelizable problem on a particular architecture: locating and extracting implicit parallelism and providing a mechanism for the parallelism to be denoted explicitly. Each has advantages and disadvantages for both the programmer and the architect.

2.3.1 Extracting implicit parallelism

There are two general approaches to extracting implicit parallelism. In the first approach, the programmer writes essentially sequential code that is translated by either the compiler, the run-time system, the architecture, or some combination into parallel code. This is essentially the approach used by out-of-order execution units on several modern microprocessors[5]. The chief advantage is that the programmer need only write sequential code. Unsurprisingly, this is also the main disadvantage. Attempting to determine which operations can safely be performed in parallel is not only difficult, but also adds substantial complexity, either in hardware[5] or software[7].

By forcing code into sequential semantics, the translational approach requires the programmer to over-specify the operations needed to solve the problem[6]. This requirement is eliminated by a second approach, often applied by functional languages such as Haskell. In this approach, the programmer is encouraged to minimize the

evaluation order dependence of code. In a purely functional languages this evaluation order independence is free. Techniques such as lazy evaluation and memoization can greatly enhance parallelization and performance[14]. However, describing some operations in a purely functional language can be cumbersome[19]. Functional languages hide the underlying architecture, even when that architecture would be useful in solving problems. It then becomes the responsibility of the compiler or run-time system to convert a functional solution into one that better exploits the machine. This problem can be as difficult as the sequential translation problem.

2.3.2 Denoting explicit parallelism

Many modern languages give the programmer the option of explicitly specifying operations that can be performed in parallel inside an otherwise sequential semantics. Examples of this approach include C*'s `poly` types and `domain` constructs[3] and Java's threading model[8].

While this approach eliminates much of the complexity that arises when attempting to extract implicit parallelism, current designs suffer from two key disadvantages.

First, the language designer is forced into choosing a particular abstract model of the machine to present to the programmer. In the case of Java, for instance, the programmer sees the machine as a Java virtual machine(VM) which can run an unlimited number of threads with a small set of priority levels[8]. This is not necessarily an optimal perspective to give the programmer. Suppose the programmer wants to map a complex function onto an array of 1024 elements. Presented with the Java thread model, she might intuitively assign a thread to calculate the result for each element. However, if the Java VM is actually running on a 4-processor machine, the overhead of context switches might make this approach undesirable. The problem gets worse when threads need to communicate. If threads are assigned amongst many processors on a distributed system, some pairs may be able to communicate with far less latency than others, but the programmer has no way to know this from the model.

The second problem is the complement of the first. Just as the programmer cannot determine how best to divide a process amongst indistinguishable threads that are

actually running on a heterogeneous system, neither can the machine determine how best to schedule threads or distribute different computations without extensive analysis either at compile-time or run-time. One Ada system described in [6] scheduled tasks off a queue by assigning to the first available processor. If two highly communicative tasks were assigned in this manner to processors with a low bandwidth or high latency interconnect, the system could grind to a halt.

A language designer might solve this problem by exposing more details of the implementation to the programmer, but too much exposure can defeat the portability and abstraction characteristics essential to a high-level language.

We need to find a reasonable abstraction that does not overburden the programmer with excessive detail, but provides enough information about the performance characteristics of the underlying architecture for an application to exploit the available hardware parallelism in a close-to-optimal way.

Chapter 3

Current Approaches

The problem of explicitly denoting parallelism and other forms of concurrency has been taken up by several language designers. We will consider a few relevant examples, analyzing the tradeoffs they make between programmability, flexibility, and performance. We will pay particular attention to features in the language that expose underlying performance characteristics of the architecture.

3.1 High Performance Fortran

3.1.1 Goals

High Performance Fortran (HPF) is a recent variant of the venerable Fortran language, designed to take advantage of advances in supercomputer architecture, particularly in the area of distributed computing and parallelism[16]. The original HPF-1 specification denotes three key areas of concern:

1. Data parallel programming

2. Top performance with non-uniform memory access

3. Code tuning for various architectures

3.1.2 Language constructs

To address these issues, a variety of language constructs were introduced.

3.1.3 Data parallel programming with FORALL

The FORALL construct deals largely with the first problem. It allows the programmer to specify an operation or set of operations to be performed in parallel, rather than using a traditional FOR loop with a sequential semantics. The compiler and/or architecture is then free to perform these operations in any order and with any degree of parallism of which it is capable.

The kind of operations performed by FORALL are by nature uncoupled. Aside from potential exception conditions (such as a divide by zero), each operation performed in parallel is independent of the others. These so-called "embarassingly parallel" problems give the implementation a great deal of flexibility.

3.1.4 Optimizing coupled operations

Not so with more tightly coupled operations that, while they may be performed in parallel, contain dependencies on other operations mediated either though reading or writing a common data space or communicating requests and responses. One general class of these problems is a producer-consumer or "pipelined" system that requires the successive calculation of several functions, each of which takes as input the output of the previous function. With a large data set, it is efficient for all stages of the computation to be performed simultaneously, each on a data item in a different stage of the pipeline.

Another parallel but coupled problem occurs in image processing. Generally, image processing algorithms are spatially local, which is to say that the processing of an individual pixel largely depends on its value and those near it, rather than on pixels further away. A good example is Photoshop's Impressionist filter, which relies on averaging the color values of adjacent pixels (among other operations) to create a blurred, Monet-like appearance.

The performance of these kinds of computations depends greatly on how efficiently their coupling can take place. If process A is computing the result of the first pipeline stage or one corner of an image and process B is computing the result of the second pipeline stage or an adjacent image section, the overall performance of the operation will depend greatly on how quickly information can pass from A to B (and vice-versa in the image processing case), as well as how quickly the processors that are performing these operations can obtain the data they require.

HPF provides a mechanism to answer this requirement (the second goal listed above), through a set of data distribition directives. The `PROCESSORS` directive allows the programmer to request an abstract rectilinear collection of processors for the execution of a set of operations, and the `DISTRIBUTE` and `ALIGN` directives to assign data to these abstract processors. The assumption is that the compiler or architecture can better allocate its physical processors to data and processes when it understands the model the programmer requires for her operations. Further, the assumption is that rectilinear arrays provide a reasonable abstraction both for the programmer to program to and the system designer to implement. While for many problems and architectures this is the case, it is not a universally true assumption. Many software engineering problems are less regular in their couplings, and certain novel and/or widely-distributed architectures (any distributed system built to operate over the Internet, for example) do not easily map to rectilinear processing arrays. However, the first assumption is definitely valid for a wide range of cases, and provides much of the basis for the new model we'll describe later.

3.1.5 Exposing the architecture

HPF goes a step farther in exposing its underlying architecture to the programmer. In addition to allocating abstract processor arrays, the programmer may also query the physical design of the underlying machine through the `NUMBER_OF_PROCESSORS` and `PROCESSOR_SHAPE` intrinsic functions. Armed with these functions, the programmer can optimize the distribution of operations in a way appropriate to the target machine. For example, when executing an image processing algorithm, there is a de-

cision to be made as to how finely to divide the image such that each section will be processed in parallel. Too many parallel processes might overwhelm an architecture, but too few will starve it. Using the `NUMBER_OF_PROCESSORS` and `PROCESSOR_SHAPE` functions, the programmer could determine the largest available set of sufficiently coupled processors, and divide the image up in such a way as to map easily to this set. The key advantage of this approach is its dynamism: the application will adapt at run-time to the architecture on which it is executed. This is another important idea we will apply later in our new model: distribution questions that are based on algorithm design can and should be made at design time, and executed at run-time. The alternative is for the compiler or programmer to attempt to guess what the underlying architecture of execution will look like, and for the architecture to attempt to guess the distribution pattern that the programmer had in mind.

HPF-1 limited its processor allocation and distribution constructs to data. The addition of an `ON` directive to the HPF-2 standard allowed processes themselve to be allocated abstractly by the programmer, rather than being determined by the compiler or run-time system. This is yet another important means for the programmer to express her intentions to the compiler and architecture, and is particularly important for software engineering-style problems that lack an easily predictable process distribution or a one-to-one mapping between processes and data sets.

Making explicit all this mapping can be difficult for a programmer, particularly when the algorithm involved does not comport itself well to a regular, geometric arrangement. In his PhD thesis describing the Connection Machine and its C* language with similar processor allocation and data distribution mechanisms as HPF, Danny Hillis includes a lengthy discussion of how traditional data structures such as lists, trees, and graphs can be mapped into rectilinear arrays for efficient processing in the Connection Machine/C* model[17]. For the software engineer who deals with even less regular structures and interactions as a matter of course, this kind of mental and programming overhead is rarely acceptable.

3.2 Software engineering languages: Java and Cilk

As a result, software engineering oriented languages offer very different mechanisms to take advantage of parallelism, traditionally in the form of threads. Threads are low-overhead "lightweight" processes that can generally share data through a common memory space. Though in the past threads were often provided as a operating system or library mechanism, they are increasingly becoming an integrated part of language syntax and semantics. Two languages which include thread semantics are Java[8] and Cilk[21].

With Cilk, the emphasis is on simplicity. A "faithful" super-set of C, Cilk adds the semantics of thread spawning and synchronization through some straightforward syntax. Spawning a new thread is simply a matter of adding a keyword in front of a function call. Synchronizing threads is similarly accomplished through a single statement, and locking is implemented by a set of provided library functions.

Java's approach is more involved, with thread creation requiring the implementation of a `Runnable` object, which is then passed to a `Thread` constructor (alternatively, the `Thread` class itself may be subclassed). As a reward for this added complexity, the programmer can provide thread-local storage, examine and interrupt threads and control their grouping and priority.

From a software engineering perspective this approach is extremely useful. Producer/consumer, master/slave, and many other designs can be implemented in a sensible fashion. However, the flexibility of this approach comes at a price. As implemented in Java, for instance, threads are inappropriate for fine-grained data parallelism: the overhead required in spawning a thread for each element of an array is usually in excess of the gains to be made by performing an operation on that array in parallel.

In fact, the flexibility provided by the Java and Cilk thread mechanisms can be a losing tradeoff for both the programmer and the language or architecture implementor. Without knowing how the programmer will rely on particular features of the threading system, the run-time system cannot make too many assumptions about how

to optimally distribute threads. In the Java model, the only information communicated by the programmer to the run-time system on spawned threads is priority level. At the same time, the programmer has no idea how efficiently the underlying architecture will spawn, schedule and execute threads. In fact, some Java implementations running on particular architectures might benefit from massive thread spawning for data-parallel operations (a hypothetical Java implementation for the Cray (nèe Tera) MTA, for example [20]), while another implementation might choke on the overhead (a dual processor Pentium II system running Windows NT, for instance).

Similarly, the programmer has no control over the distribution of threads. Threads which are tightly coupled, such as a producer and consumer thread, will perform optimally when the connection between them is fast and the two processors on which they execute are comparable in speed. If the producer thread, for example, is on a much faster processor, it will quickly produce too much data and either stall or overflow the buffer of the consumer thread, depending on the application's design. In either event, productive work comes to a halt.

In traditional threading models, the application can express little information on data and process parallelism to the compiler or run-time system, while at the same time, the run-time system provides little useful information to the programmer on how its underlying architecture might be optimally exploited. This lack of information exchange severely limits both the kinds of problems suitably addressed by thread-based languages and the range of architectures on which those languages can be implemented. A Java implementation, for instance, could not usefully scale to a distributed architecture run over the Internet, since the programmer could not be certain that tightly coupled threads would not end up behind 14400 baud modems in Helsinki and Caracas. Nor could a massively data-parallel problem effectively be implemented in Java with the hope of efficient execution on multiple architectures. The programmer would have no idea the correct number of threads to spawn (10? 1,000? 1,000,000?) to properly distribute the processing task, and the run-time is specification-bound to spawn exactly that number of threads.

In both systems, the underlying implementation has control of how and where

threads are executed and scheduled. Often, implementations have heuristics which attempt to guess which threads should be given priority in the future, as well as which threads are most likely to communicate and thus should be placed on either the same or closely coupled processors. However, this information is often obvious to the programmer as a consequence of his design. This is not always the case, as data dependencies can determine which threads require a higher priority or faster communication for optimal performance.

We'll turn now to a model that attempts to separate decisions affecting performance from those that can be made at design time as a function of the underlying architecture, to those that can only be made at run-time based on data dependent operations. With this model we hope to combine the flexibility of the threading approach with the dynamic adaptability provided by the architectural exposure of languages like HPF.

Chapter 4

Self-Distributing Computation

Self-distributing computation is based on a fundamental principle:

Process and data distribution decisions which can be anticipated and reasoned about at design time should be made at design time, while decisions depending on run-time information should be made at run-time.

4.1 Choosing an abstraction

As we have seen, this seemingly obvious maxim is not universally applied, particularly in software engineering-focused languages such as Java. The underlying architecture has characteristics which can severely impact the performance of different distribution patterns, yet these characteristics are not exposed by the Java Virtual Machine abstraction.

Here we will argue for an abstraction which, like High Performance Fortran, does expose performance characteristics to the application, but does so in a manner more conducive to the irregular design of software engineering problems, rather than restricting the architectural representation to rectilinear arrays.

4.1.1 The graph representation

We choose to generalize the architectural representation to that of a directed graph. This has two key advantages. First, it allows the representation of much more complex interconnect and network topologies, such as fat trees, rings or tori. In fact, HPF recommends that implementors support directives to denote particular topologies in interconnect, but the use of a directed graph obviates the need for an ad-hoc approach. Second, it allows attributes to be associated with each node and edge that allow the architecture to express its internal heterogeneity, whether it be processors of differing clock rates, connections with various latency, bandwidth and buffering characteristics or other factors that could impact performance.

The ideal representation of this graph (as well as its nodes and edges) would be polymorphic. This would maxmize both backward and upwards compatibility. The run-time system on a given platform could provide a graph type with extensive details about its unique features, but one that is compatible with a simpler, more general type. Programmers could then choose to exploit the more specific type if the run-time system can provide it and their solution could benefit from the additional data, while leaving the more general type as a fall-back in the event the underlying architecture changes and can no longer provide the detailed type. While a full OO-style polymorphic type may not be practical in an efficient run-time system, one can be simulated with an associative map. Certain general properties, such as `number-of-processor-nodes` and `network-latency` could be guaranteed to appear in the mapping, while more platform-specific properties, such as `processor-branch-mispredict-penalty` could appear optionally, and applications could adjust to their presence or absence accordingly.

4.1.2 Controlling distribution and consolidation

Of course, determining the run-time information to be presented to the program only addresses half of the design problem. The other issue concerns the protocol by which the graph is presented to the program by the architecture and the application suggests

(or commands!) a distribution pattern. There are two general approaches we might take.

The first is the HPF approach, what we might call application-driven. In the application-driven approach, at any point the application can request information from the run-time system and request the allocation of processors and the distribution of processes and data amongst them. This might seem ideal from a programmer's point of view. She will have complete control of the distribution process (or at least the virtual distribution process), and can deterministically reason about the evolution of her processes and data.

There are two unfortunately consequences to this approach. For one, the requirement that the programmer trigger the process of distribution can lead to a tangling of two distinct aspects of the code: the actual implementation of the parallel or distributed algorithm being executed and the code required to find and create an optimal distribution. In some ways the distribution policy of code is what the Aspect Oriented Programming community terms a "cross-cutting concern"[12]. A good example is an image processing algorithm. The domain of the algorithm might merely be pixels on an image, yet in order to decide effectively in the application-driven approach whether the processing of two segments of the image might be best split amongst two processors, the application's code would need to perform run-time information queries in the middle of an image processing routine.

The second consequence of a application-driven approach is that programmers often cannot (or do not!) anticipate changes in the run-time situation that might occur due to numerous events either inside or outside the programmer's purview. Data dependent operations can lead to suboptimal performance that the programmer may not have anticipated.

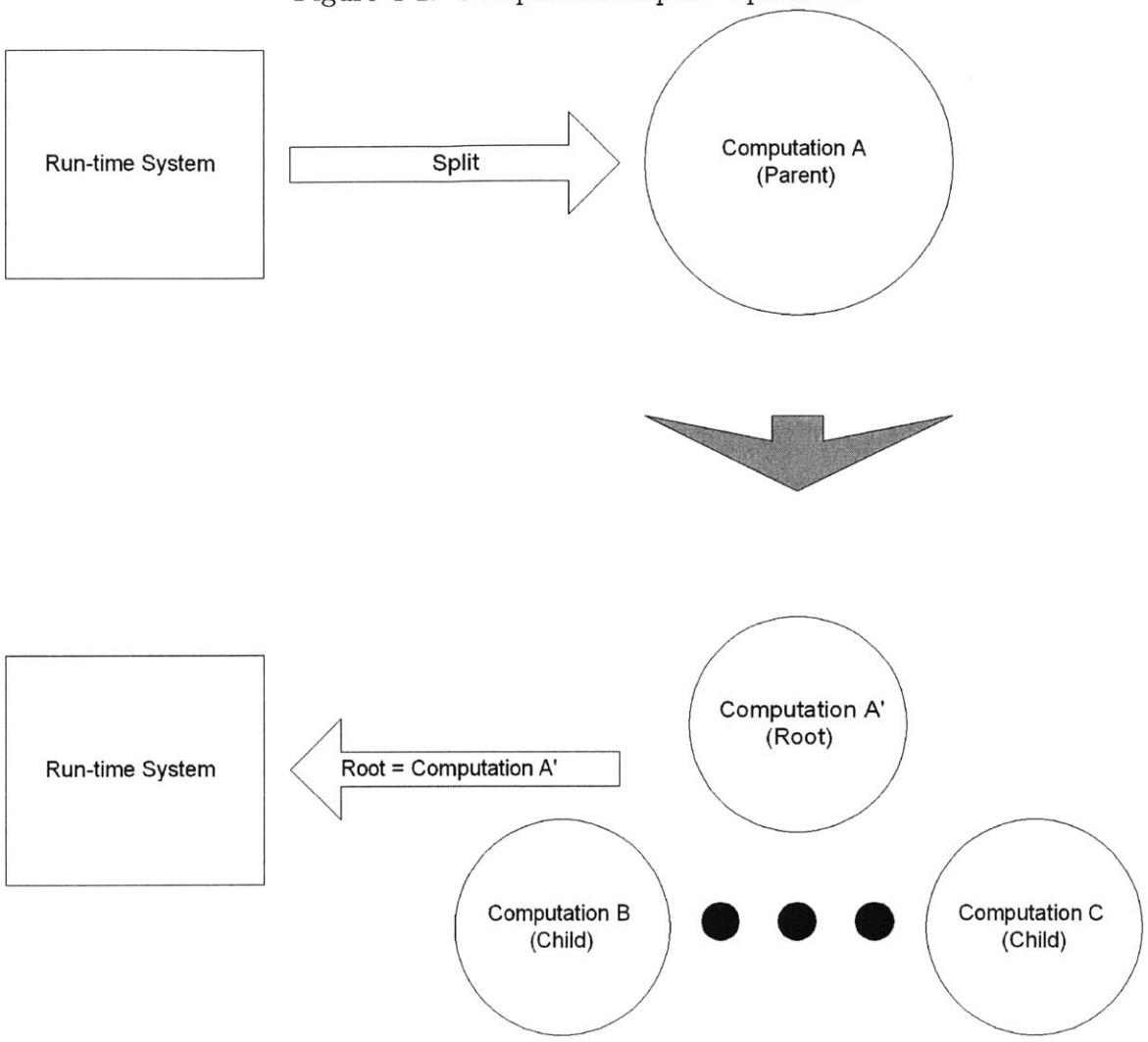
To return to the image processing example, suppose the particular algorithm being implemented requires finding a fixed point of multiple iterations, each iteration involving a specific manipulation of adjacent pixels. If finding the fixed point of the algorithm along the boundary of two segments will require far more than the usual number of iterations, and those two segments are assigned to separate processors,

the algorithm could be slowed substantially by the unexpectedly high communication traffic between the two processors. The programmer has no way of anticipating this problem, as in the usual case the two segments would be minimally coupled and the distribution would result in a performance gain. However it could be apparent to the run-time system that whatever is actually going on between the two processors, their performance is being limited by the communication between them. One easy way for the run-time system to make this determination would be to compare the processor's cycle utilization with its network utilization. It then would become obvious that placing both processes on a single processor would improve performance, as on-chip bandwidth and latency is usually at least an order of magnitude better than inter-processor bandwidth and latency regardless of interconnect.

Similarly, factors external to the application's tasks could affect when it would be globally optimal to reallocate processors and redistribute data. In a multi-programmed system, a new application might be started, or another application might be given higher or lower priority and a correspondingly greater or smaller allocation of processors for its use. The physical layout of the machine could change, with processors hotswapped in or out, or new networked machines added to a distributed system. Of course, the application is likely to be completely unaware of these changes unless it meticulously checks for updates in the run-time situation, which, in addition to being a burden on the programmer, may also result in lower performance if these checks occur in a "polling" fashion even when the run-time situation has not changed. Without this co-operation from the programmer, the run-time system faces two choices: it can ruthlessly reallocate processors and change the virtual-to-physical processor mapping, perhaps in a way that severely affects the performance of the application, or it can put off any changes until the application naturally reallocates and redistributes or finishes. Neither approach is particular appealing.

The alternative, then, is to allow the architecture and its run-time system to trigger the distribution and consolidation of processes. From the application's perspective, these triggers will occur asynchronously, interrupting the algorithm mid-flight. If the run-time system could make arbitrary requests, it would greatly burden the

Figure 4-1: Computation `split` operation



programmer. She would be forced to take into account a huge array of scenarios, generating correct behavior for each and would likely be unable to concentrate on optimizing the likely cases. Therefore it makes sense to restrict as much as possible the potential distribution and consolidation actions of the run-time system.

The approach taken by the SDC model relies on the notion of a process tree. Each node and leaf of the process tree corresponds to a computation. Computations are best imagined as process objects which contain sufficient metainformation and operations to respond to collection and distribution requests from the run-time system. Initially, each application is represented by a single computation being executed on a single (virtual) processing node.

The run-time system initiates distribution by executing a `split` operation on a computation (see Figure 4-1). Included in the `split` request is a graph representation of the kind previously discussed. The run-time system is free to decide (again, as described above) not only how detailed the representation is, but also what portion of the underlying architecture to include. Higher priority tasks, for instance, might be exposed to a larger collection of virtual processors so that they could be distributed sensibly among larger collections of physical processors. We will discuss other possible heuristics for this decision later.

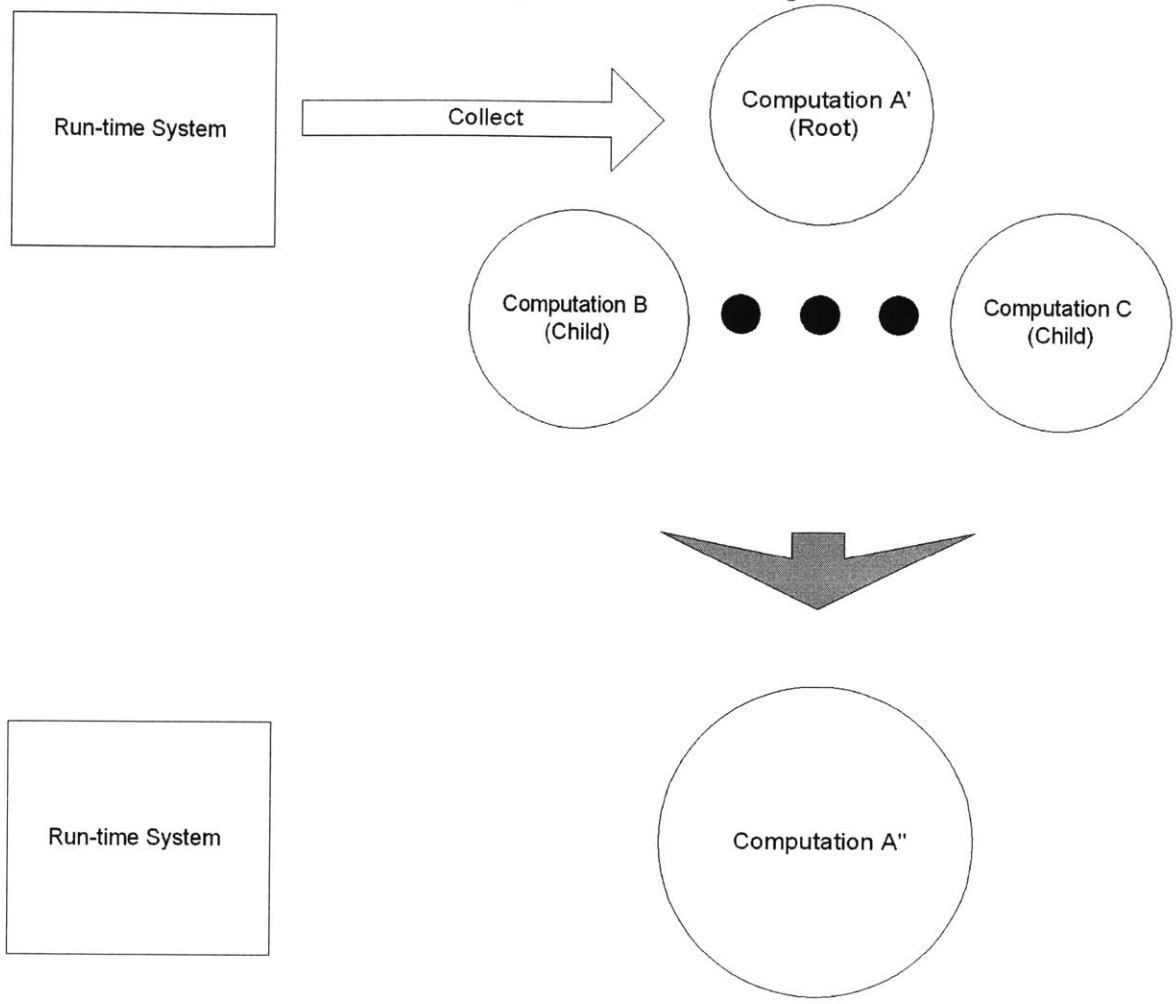
Once it receives the `split` request, the computation must, on the basis of introspection as well as examination of the representation graph, create and configure a set of new computations to perform its original task in a parallel or distributed fashion. We will refer to the original computation as the parent computation and these new computations as child computations. It responds to the run-time system with a computation map, mapping nodes in the representation graph to the child computations.

In addition, the computation specifies one of the child computations to act as a root. It is this computation which must be prepared to respond to a `collect` request from the run-time system. Upon receiving a `collect` request, the root computation must generate a single computation which can perform the task of its parent computation (see Figure 4-2). The behaviour of a non-root computation upon receiving a `collect` request is undefined.

These requests can be made recursively, resulting in a tree structure. The growth of the tree is triggered by the run-time system and controlled and implemented by the application, while collapse of the tree is triggered and controlled by the run-time system, and merely implemented by the application.

From an implementation perspective, there is a trivial means for a computation to operate: a `split` request can result in a single computation, identified as the root, that is identical to the parent computation. This pseudo-root can respond to a `collect` request with itself. The implementation of more complex designs depends on details of the SDC implementation. An implementation might, for instance, allow

Figure 4-2: Computation `collect` operation



a root computation some fixed period of time to generate the collected computation before it is free to remove the other child processes. The simple Java-based implementation discussed in the next chapter used internal state of the root computation alone to generate the collected computation. This approach resulted in a simple implementation, but limited the flexibility of distribution and made it difficult to extract fine-grained parallelism out of some algorithms. We wil discuss this tradeoff more later, as it is of crucial importance to the viability of the SDC approach.

4.2 Advantages of the model

The SDC model overcomes many of the failings of traditional thread-based approaches by providing an abstraction that does not conceal performance information that is critical to the optimal execution of a parallel or distributed program. It also cleanly separates distribution policy concerns from algorithmic execution. It frees programmers to anticipate design optimizations at design time. Many thread distribution and scheduling systems implemented by operating systems or run-time systems, for example, must make a blind tradeoff between the time required for analysis and the benefits of a more optimized distribution. Too much time spent on analysis limits available "productive" cycles, and may be wasted on truly unpredictable data dependent algorithms that would be best distributed at random. Of course the programmer is usually in a position to understand his algorithm and its requirements at design time, and can choose the complexity of the distribution algorithm accordingly. Allowing the application programmer control of these kinds of decisions was a key lesson of the Exokernel project when applied to operating systems[9], and it is unsurprising that it can be applied to distributed and parallel programming systems.

Correspondingly, it removes much of the burden of dealing with pathological and exceptional cases from the run-time system. The failure of a processing node can always be dealt with by collecting at a higher-level root computation, so long as that computation's collection mechanism recognizes the possibility of node failure. Similarly, dynamic changes in network configuration or other resource availablity can be exploited by both the run-time system to accomodate its own policies and by programs to optimally use resources available.

4.3 Limitations of the model

The SDC model does have several limitations. While representing available processing and communication resources via a graph is a more general and flexible approach than rectilinear arrays, it is difficult to represent "parameterized" conditions, such

as network connection B's performance being dependent on connection A's load–as might easily occur in a switched network–without extending the model beyond recognition. However, there are advantages to the "first-order" nature of the graph representation, as it lends itself to simple analysis and distribution heuristics, e.g. "Choose the pair of processors with the least-latency interconnect and map these two communication-bound computations to them." The run-time system might consider "second-order" effects when it performs the physical mapping of these computations, avoiding a distribution that would belie the indications of the graph representation.

Another difficulty arises from the asynchronous nature of distribution and collection requests from the run-time system to the running computation. To implement a sensible distribution or collection policy, the request handler must be able to introspect the current computation or set of child computations. However, given the potentially inconsistent state of a computation at the time of a request, either the request processor must make a "safe" and potentially suboptimal decision, or wait until the computation reaches a consistent state to perform the necessary analysis. Due to the interrupt-like nature of these requests, it is uncertain whether the run-time system could allow the computation unlimited time and access to resources in order to make its determination. One solution to this problem is to increase the metainformation the computation makes available as it proceeds with its algorithm, so that distribution and collection requests can be processed immediately with more optimal results. Another approach, used by [9] would be to limit the expressive power of the request handlers, allowing the run-time system to have guarantees about their required processing time.

The Java implementation we'll discuss next uses the former approach, and we will pay particular attention to how this affects the flexibility and programmability of the model.

Chapter 5

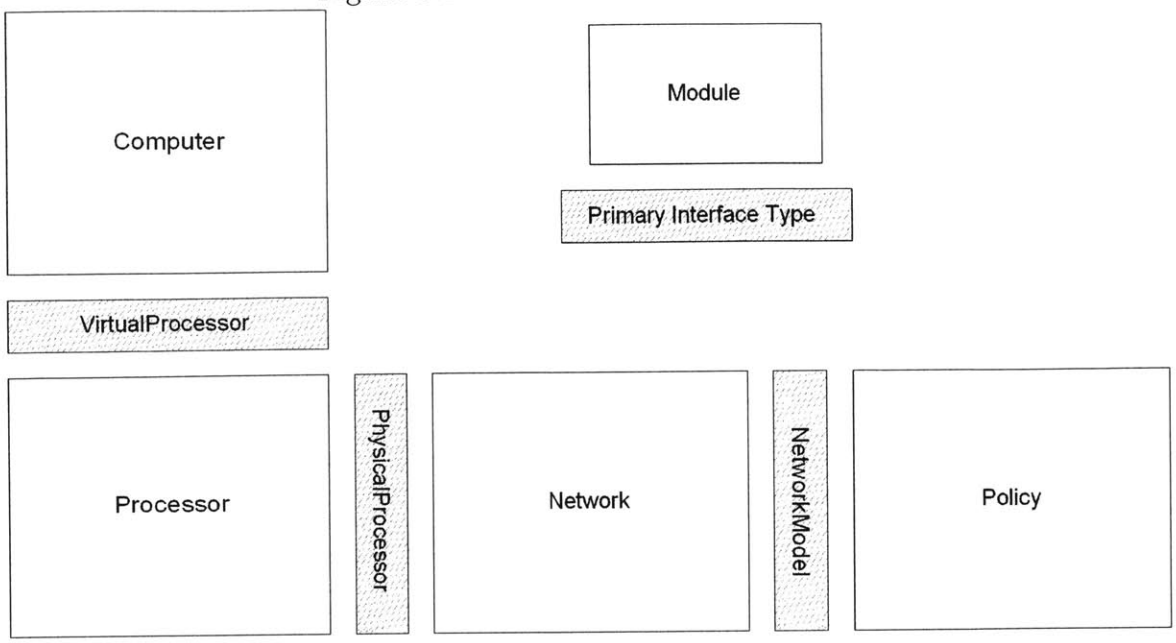
The Mimoid Implementation

To explore the viability of the SDC model, a prototype was created in the Java language. Called Mimoid, it presented an opportunity to examine both architectural and run-time implementation issues as well as programmability concerns. While not particularly complex, the system and algorithm prototypes involve approximately 2,500 lines of code.

5.1 Mimoid structure

The prototype system is built around four modules, each with one or two key interfaces (see Figure 5-1.). The `Processor` module concerns the actual execution of computations, presenting both a virtual processor interface to the computation itself and a physical processor interface to the rest of the system. The `Network` module represents the interconnect between the processor nodes, with interfaces both to these nodes and to the distribution policy. The `Policy` module contains types for the run-time graph representation as well as types which allow the specification of rules for distributing and collecting computations. The test computations themselves take advantage of a `Computer` module, which simplifies the test programmer's interface by taking care of much of the administrative work in creating a network of nodes, assigning initial processes, and bootstrapping the system.

Figure 5-1: Mimoid structural overview




```
public interface Computation extends Runnable {
    public VirtualProcessor getVirtualProcessor();
    public void setVirtualProcessor(VirtualProcessor vp);
    public void run();
    public boolean isRunning();
    public boolean isDone();
    public ComputationMap split(TargetGraph tg);
    public Computation merge(NodeDescriptor nd);
}
```

Listing 1: Computation.java

5.2 Computations

The critical type from the programmer's perspective is the `Computation` (see Listing 1). Much like the Java `Runnable` interface (which `Computation` extends for implementation reasons), a `Computation` is designed to encapsulate a locus of control that can be executed by a thread (or in our case, a processing node). In addition to the `run()` method, the `Computation` must support the split and collect actions discussed previously. This is done through the implementation of the `split()` and `merge()` methods.

The `split()` method is parameterized on a `TargetGraph` object (see Listing 2),

```
public interface TargetGraph {
    public Set getNodeDescriptors();
    public Set getConnectionDescriptors();
    public NodeDescriptor getCurrent();
}
```

Listing 2: TargetGraph.java

```
public interface ComputationMap {
    public Computation map(NodeDescriptor n);
    public NodeDescriptor getRoot();
}
```

Listing 3: ComputationMap.java

which is the run-time graph representation discussed earlier. It is composed of two sets, a set of `NodeDescriptors` which correspond to processing nodes in the underlying run-time representation, and `ConnectionDescriptors`, which correspond to the interconnect between nodes. It also contains a reference to a `NodeDescriptor` corresponding to the node currently occupied by the parent `Computation`. This allows the `Computation` distributed to this node to rely on state information created there by the parent `Computation`.

The `split()` method returns a `ComputationMap` (see Listing 3) which contains both a mapping between `NodeDescriptors` and new `Computations`, as well as a reference to a `NodeDescriptor` of the location of the new root `Computation`, which is usually, but is not required to be, the location of the original parent.

The collect operation performed by the `merge()` method is parameterized on a single `NodeDescriptor`. This parameter represents the future home of the collected `Computation`, and during the collect operation, the root `Computation` can derive information from this descriptor that would allow it to optimally combine the current child `Computations`. For example, a `Computation` designed to perform a search might choose an algorithm for the reconstituted data that fits the performance characteristics of the node on which it will be performed. The result of the `merge()` method is a single `Computation`, that, under the SDC model, will perform the same function as the family of `Computations` of which this call was made on the root.

```
public interface VirtualProcessor {
    public int getMemSize();
    public void writeMem(int addr, int value);
    public int readMem(int addr);
    public void sendMessage(Message m);
    public boolean isMessageWaiting();
    public VirtualProcessorAddress getAddress();
    public Message receiveMessage();
    public void commitMessage(int addr);
}
```

Listing 4: VirtualProcessor.java

5.3 Virtual processors

In order to best analyze the operation of test processes in this model, the Mimoid architecture provides a single, narrow interface through which `Computations` can perform significant (i.e. visible to the external world) work. This interface is the `VirtualProcessor` (see Listing 4). Far simpler than any actual processor, the `VirtualProcessor` nonetheless has sufficient operations availble to perform any computationally signficant task in a way that is easily instrumentable for control and analysis.

The `VirtualProcessor` interface provides uniform access to a word-addressable memory array, as well as a port for sending messages to other processors and a queue of received messages. The memory interface is simplicity itself. The `getMemSize()` method returns the size of the memory in words. The `readMem()` method retrieves a word value from an address in memory, while the `writeMem()` method stores a word value into an address in memory.

Message passing is slightly more complex. Each `VirtualProcessor` has an address, retrievable by the `Computation` through a `getAddress()` call. The address has a numeric representation that can be stored in a word for either future reference or to pass to another `Computation` for its use. Messages themselves are created as records containing the fields described in Table 5.1. Messages are sent through a `sendMessage()` method.

Table 5.1: Inter-computation Message Fields

Field Name	Field Type	Description
DestinationAddress	VirtualProcessorAddress	Identifies the virtual processor to receive this message
BufferStart	int	Points to the location of a buffer in the source processor's memory to be used for the message content
BufferLength	int	Specifies the length of the buffer
MessageID	int	A tag for communication protocol use (e.g. a sequence number)

Upon reaching the destination `VirtualProcessor`, that processor will add it to an internal queue. The `Computation` running on that processor can query the status of that queue with the `isMessageWaiting()` method, and retrieve the top message from the queue with the `receiveMessage()` method. Retrieving a message off the queue does not place its content into the `VirtualProcessor`'s `Computation`-accessible memory array. That requires a subsequent `commitMessage()` method call, which will store the most recently received message content into local memory at an address specified by a parameter to the method. Only the most recently received message can be committed, which allows the `VirtualProcessor` implementation to discard all old messages: either a message is committed after it is received, in which case it can be discarded as delivered, or it will not be committed before another message is received, in which case its content is inaccessible to the `Computation` and can be discarded.

5.4 Physical processors

From the underlying system's perspective, processors are represented through the `PhysicalProcessor` interface (see Listing 5). This interface provides the system with the ability to start and stop a `PhysicalProcessor`, as well as execute a clock "tick" through the `tick()` method. This allows the prototype to step each processor's execution and perform both administrative work (such as reporting activity to the simulator user) and policy work (such as determining whether to issue a `split`

```
public interface PhysicalProcessor extends Port {
    public Computation getComputation();
    public void run();
    public void halt();
    public void tick();
}

public interface Port {
    public void receiveMessage(NetworkMessage m);
}
```

Listing 5: PhysicalProcessor.java and Port.java

request).

A `PhysicalProcessor` must also provide a `Port` interface, which allows messages from the network (typed, unsurprisingly, as `NetworkMessages`) to be sent to the processor. It is through these messages, discussed further later, that the Mimoid implementation implements process distribution and collection as well as interprocess communcation.

The current `PhysicalProcessor` interface also provides direct access to the `Computation` object executing on it. This is an unfortunate violation of encapsulation, as not only does it expose what should be the province of the virtual architecture, but also in so doing it prevents multiple `Computations` from executing on the same `PhysicalProcessor` in a context-switched fashion. This is a failing of the current interface between the `Computation` and the `VirtualProcessor`, which does not allow the `Computation` to express to the `VirtualProcessor` that it has completed, which is necessary knowledge for the simulator to determine correctly when to terminate. This wart should be removed in the next version.

5.5 The network model

The network itself is simulated through a `NetworkModel` interface (see Listing 6). The `NetworkModel` interface is suprisingly simple, though a number of complex implementations could be imagined. Essentially it provides an operation for the system to note

```
public interface NetworkModel {
    public void insertMessage(NetworkMessage nm, NodeAddress from);
    public void tick();
    public Set getDeliveredMessages();
    public TargetGraph getRepresentation();
    public boolean messagesWaiting();
}
```

Listing 6: NetworkModel.java

```
public interface DistributionPolicy {
    public Set chooseSplits(TargetGraph graph, Set nodes);
    public Set chooseMerges(TargetGraph graph, Set nodes);
}
```

Listing 7: DistributionPolicy.java

when and where `NetworkMessages` enter the network (`insertMessage()`), an operation to denote the passage of time (`tick()`), and operations to locate and retrieve messages that have reached their destinations (`messagesWaiting()` and `getDeliveredMessages()`). In addition, the `NetworkModel` is the logical place to originate a representation of the current state of the system, so a `NetworkModel` can provide, through the `getRepresentation()` method, a `TargetGraph` that can form the basis for the `TargetGraphs` which are included in `split` requests.

5.6 Distribution policy

The triggering of distribution and collection requests is controlled by implementations of another interface, `DistributionPolicy` (see Listing 7). `DistributionPolicy` contains two methods, one for selecting splits and the other for selecting merges. Each takes a current representation of the state of the system in the form of a `TargetGraph` and a set of `Nodes`. The sets returned by each method contain either `Distribution` or `Merge` objects, depending on the method called. `Distribution` objects contain an address of the `Node` object containing the `Computation` to issue the `split` request to, as well as the `TargetGraph` to include in the request. This may or may not be the same `TargetGraph` passed to the policy, since the policy may specify the allocation of

only a subset of the available processing nodes and network connections to a particular Computation. Merge objects contain the address of the Node object containing the Computation to issue the collect request to, as well as the NodeDescriptor describing the node where the resulting Computation will be placed.

5.7 Putting the pieces together

Perhaps not surprisingly, it was discovered that the easiest way to implement the VirtualProcessor and PhysicalProcessor interfaces was through a single class, the BasicProcessor. The BasicProcessor implements the memory array functionality of the VirtualProcessor interface through an array of ints. The sendMessage() method involves repackaging a virtual Message object as a NetworkMessage and passing it along through a Port connected to the BasicProcessor at construction. NetworkMessages received through the PhysicalProcessor receiveMessage() method are examined for their type. If they are simple inter-process messages their virtual Message objects are unpackaged and placed in a queue for access by the Computation through the VirtualProcessor interface. Policy messages (such as split and collect requests) are dealt with in a manner described below.

A BasicProcessor also contains a reference to a Computation object, as well as a Thread object to execute the Computation. This means that the actions of Mimoid Computations are specified in the Java language. An alternative would have been to describe the execution of Computations in terms of some simple interpreted language, which could be executed step-wise by the BasicProcessor. The consequences of this approach would have been the obvious requirement of writing an interpreter, as well as forcing test Computations to be written in a rather restricted language. Using a Java thread as the "interpreter" requires some clever management overhead, but overall results in a simpler test environment, ensuring that programming issues are the responsiblity of the SDC model, rather than of a hobbled implementation language. Specifically, VirtualProcessor methods that will be called by the Computation's thread must be instrumented to allow only a single operation to be performed during

each PhysicalProcessor "tick." This is accomplished through use of the (deprecated, but still supported) Thread suspend() and resume() methods, which, while deadlock-prone in the general case, are quite safe and useful here.

The processing of split and collect requests is handled through the messaging system. When the BasicProcessor receives a NetworkMessage through its PhysicalProcessor interface, it checks to see if the NetworkMessage is one of four special types.

If it is a SplitMessage, it represents a split request, and contains the necessary TargetGraph. The BasicProcessor extracts this TargetGraph and calls its Computation's split() method with the TargetGraph as a parameter. It then analyzes the returned ComputationMap. For each mapped Computation, it constructs another special NetworkMessage of the DistributeMessage type. The DistributeMessage type contains a representation of the new child Computation, and this message is sent to the processor on the node corresponding to its NodeDescriptor in the ComputationMap. In addition, a RootMessage is sent to a special replyTo address included in the SplitMessage. This informs whichever entity triggered the distribution that a particular node contains the root Computation, and that future collect requests should be directed there.

If a DistributeMessage is received by a BasicProcessor, it simply extracts the contained Computation and replaces its current Computation (if any) with the new Computation, severing the old Computation's reference to the VirtualProcessor interface. Although the current Computation might still be executing, without a reference to a VirtualProcessor, when it next attempts to read or write memory or send or receive a message it will throw a NullPointerException and terminate.

The converse of the SplitMessage is the MergeMessage, which represents a collect request. A NodeDescriptor is referenced in a SplitMessage, and this descriptor is extracted and passed to the current Computation's merge() method. The Computation returns another Computation result, which is packaged in a DistributeMessage and entered into the network.

Finally, a BasicNetworkMessage represents an inter-process message and is packaged into a Message object and added to the VirtualProcessor's queue as discussed

above.

The chief consequence of the `VirtualProcessor` calling methods of the `Computation` object to perform distribution and collection is that the `Computation` itself cannot issue commands to the `VirtualProcessor` (such as reading and writing memory or sending messages) when it is requested to `split` or `collect`. This means that `Computation` objects must contain sufficient internal state to perform these operations in a sensible fashion. However, it also allows the run-time system reliable control over exactly when `Computations` distribute and collect themselves.

The policy, network, processing nodes and initial computation(s) are connected and managed by the `Computer` class. Given a mapping between nodes and their addresses, a set of `Computations`, a `NetworkModel` and a `DistributionPolicy`, the `Computer` starts the processors and begins sending `tick()` requests to each processor and the `NetworkModel`, delivering `NetworkMessages` to processors as the `NetworkModel` indicates they should be. After each tick it presents the current run-time situation to the `DistributionPolicy` by retrieving the `TargetGraph` representation from the `NetworkModel`. The `DistributionPolicy` responds with sets of `Distribution` and `Merge` objects, each of which is packaged by the `Computer` into an appropriate `SplitMessage` or `MergeMessage` and inserted into the network. It monitors each `Computation` for its completion, and when all `Computations` on all processors are finished, the simulation is terminated and a total tick count returned to the user.

Some simple algorithms, including QuickSort[18] and Conway's Game of Life[15], were implemented and tested with a variety of network models and distribution policies. We'll discuss some of the conclusions reached from this work in the next chapter.

Chapter 6

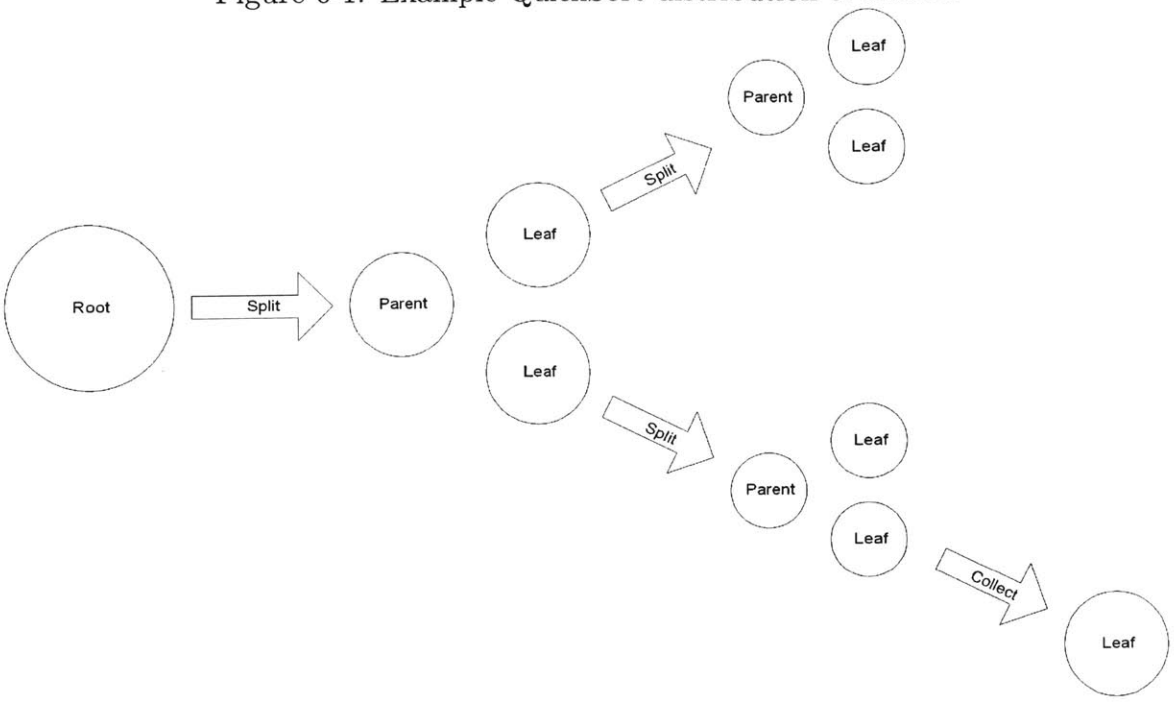
Lessons Learned

From implementing and testing Mimoid, several insights into the SDC model were gained that deserve further exploration. We will examine these insights in the context of a QuickSort program implemented on Mimoid.

6.1 The QuickSort implementation

The Mimoid QuickSort implementation relies on three distinct types of computation, each of which can be distributed to any processor on the system. Leaf computations are simply responsible for applying the entire QuickSort algorithm to a chunk of data they are sent from a Parent computation. Parent computations perform only the partition stage of the QuickSort on data they receive, separating those elements that are higher and lower than a pivot value and transmitting each set (higher and lower) to one of two Leaf computations. Finally a special kind of Parent, the Root, has the data to be sorted initially available in its processor's memory through a bootstrapping routine. Results computed by Leaf nodes are retransmitted up through the Parent nodes until they reach the Root and the computation is completed.

Figure 6-1: Example QuickSort distribution evolution



6.2 Distribution as protocol

Relying on the run-time system to trigger distribution and collection transforms the distribution problem into what is fundamentally a protocol problem. This result is similar to the shift in thinking required in the transition from thread-based concurrency to asynchronous programming. Each computation is essentially a pair of state machines, one encapsulated in the other. The internal state machine represents the underlying computational work to be done, acting on data read from memory and input or messages from other computations. The external state machine responds to `split` and `collect` messages from the system, maintaining both its own indepedent state, as well as responding to state changes of the internal state machine.

In the case of the QuickSort implementation, a Leaf computation's internal state is the array it is required to sort. It exposes to the external state machine a flag which indicates whether or not it has completed its work. The external state machine chooses its response to `split` requests based on this flag. If the work is not completed, the Leaf responds to a `split` by producing a Parent computation mapped

to the original node and two Leaf nodes to perform QuickSorts on the partitioned original array. However if the work has completed, the `split` request results in a reproduction of the original Leaf. The Root computation responds to `split` requests in the same way. See Figure 6-1 for an example of how this distribution could evolve. Considering the startup costs required to perform a `split` in this manner, it might be advantageous for the internal state machine to expose a measure of its progress rather than a boolean flag of completion. This would allow the external state machine to determine whether the performance of the `split` would require more time and resources than simply finishing the computation at the Leaf. However, even without this functionality, on a homogenous network with semi-frequent `split` requests, the distributed QuickSort was able to complete in substantially (30-50% depending on the number of splits) fewer ticks than a non-distributed version, despite the redundant work required by this straightforward (and rather naive) distribution protocol.

6.3 Separating policy from algorithm

Separating out the distribution scheme of a program as a protocol has substantial advantages. Most importantly, it abstracts many details of the process being performed. Any algorithm involving dividing a work load in two and processing each half independently could use the same protocol as the QuickSort implementation, literally without modification. Although as it is currently designed, Mimoid does not paramaterize the distribution protocol as an abstract data type, such an approach would be extremely plausible. In fact, `BasicComputation`, an abstract implementation of the `Computation` interface actually provides a default "worst-case" protocol which proved adequate for a range of simple test computations, including a network Ping-Pong exchange, the included listing of which (see Listing 8) illustrates the ease with which applications can be written for the Mimoid system.

Program distribution protocols covering a large range of complexity can be imagined. In the case of indepedent worker threads servicing, for example, web page requests, a trivial distribution protocol would be to create some number of new worker

```
public class Ping extends BasicComputation {
...
public void run() {
   isRunning = true;
   for(int i=0; i < 10; ++i) {
      BasicMessage ping = new BasicMessage(dest, 0, 0, 0);
      getVirtualProcessor().sendMessage(ping);
      System.out.println("Ping! on VP: " + getVirtualProcessor());
   }
   isRunning = false;
}
...
}

public class Pong extends BasicComputation {
...
public void run() {
   isRunning = true;
   for(int i=0; i < 10; ++i) {
      getVirtualProcessor().receiveMessage();
      System.out.println("Pong! on VP: " + getVirtualProcessor());
   }
   isRunning = false;
}
...
}
```

Listing 8: Ping.java and Pong.java

threads on each split request, and remove them on a collect request. More interdependent relationships, such as producer-consumer, require more complex protocols.

The conversion of process to protocol is also apparent in the run-time mechanism for triggering `split` and `collect` requests. Here again policy can be abstracted from details of implementation, both of the underlying architecture and the running processes. In this case, the Mimoid architecture directly supports this approach, as `DistributionPolicy` objects can be plugged into the system dynamically. This approach was useful in testing, as it was trivial to transition from a simple policy with only a single `split` to a more dynamic one which sent `split` requests to processors performing excessive memory accesses, and `collect` requests to processors engaging

mainly in network traffic. Here again we see the importance of state exposure, in this case from the processor and network to the run-time distribution policy.

6.4 Unaddressed issues

Some issues that could arise in real-world implementations of an SDC model were either impossible or impractical to address in the Mimoid implementation. While none seem insurmountable, they should be discussed further.

6.4.1 Virtual to physical mapping

One issue largely unaddressed by the Mimoid implementation is the question of virtual to physical mapping of processors and interconnect. Because of the degree of expressiveness available in the SDC model, it would be tempting to simply use a one-to-one mapping of processing nodes and interconnect in the representation graph to physical processors and interconnect. While this approach is straightforward and certainly feasible, experience developing sample applications such as the distributed QuickSort indicate that it will probably not be optimal. Separating out different structural roles for computations, as in QuickSort with the Root, Parent and Leaf computations, leads to some computations serving as infrastructure and requiring few resources. Certainly a Parent computation which spends most of its time waiting for response messages from its children does not require the full resouces of a processor. It might be sensible, then, for a physical architecture to present a single processor capable of supporting multiple contexts as several nodes of fractional computational power but fast interconnect. On the other hand, this kind of representation would be suboptimal for a computation which required the full resources of the processor. These limitations on the representation graph will be discussed more in the next section.

6.4.2 Addressing

Finally, a major issue for any implementation of SDC involving a non-global address space will be inter-process addressing. Because of the unpredictable nature (from the programmer's perspective) of `split` and `collect` requests, it is difficult to know precisely where a particular computation will be executing or which computation will own a particular piece of data. This difficulty can be overcome through careful inter-process communication protocols. This approach was successful with distributed QuickSort, where each child could wait for a parent to send it the required data and could be assured that the parent would not change locations before it had completed its work. However more flexibility would be available with a more convenient virtual addressing scheme that would allow computations to have a single address regardless of their node location.

Chapter 7

Future Directions and Potential Applications

7.1 Future directions

As we discussed in the previous chapter, there are several implementation issues that deserve further exploration. In addition, much more work could be done in designing and analyzing both application and run-time distribution policies.

7.1.1 Improving the graph representation

The key implementation question, and also the chief factor in the design of distribution policies, is what information to include in the graph representation created and presented to the application by the run-time system. For simple distribution schemes that produce computations which are not communication-dependent, a set of nodes without interconnect would be sufficient, and indeed early versions of the distributed QuickSort used just this sort of "graph." Essentially this representation is identical to the integer result of the HPF `number_of_processors` function. Obviously computation distributions with non-trivial communication requirements can benefit from knowing something about available interconnect. While some interconnect schemes are amenable to characterization through static figures–bandwidth in megabits or

gigabits/second, latency in nanoseconds–these figures are often constantly changing, dependent on both how the current application uses the interconnect, as well as the influence of outside factors such as simultaneously running applications or external traffic.

This presents a challenge to the SDC model, as it must be determined how best to take a static "snapshot" of this data that will be useful to the application in determining its distribution. On the other hand, it also demonstrates an advantage of an asychronous, run-time driven approach over the application-driven approach of such languages as HPF, as the SDC model ensures that applications will have the opportunity to respond to changing circumstances, rather than locking them into a single allocation scheme or forcing them to constantly check for situation changes.

It is unclear how much more useful information could be included in the run-time representation. Characterizing the performance of processors in a heterogeneous system, for example, is notoriously difficult, with MHz, MFLOPS, or even SPEC figures often insufficient to communicate actual performance potential for a given problem. On the other hand, even a foggy performance number is probably better than none at all, as order-of-magnitude differences in a highly heterogeneous network could certainly be significant to an application.

7.1.2 Exploring application distribution policies

What kinds of distribution policies might apply to different types of application is an open question. Certainly much research has been done into parallelizing common (and not-so-common) algorithms. Less work has been done on what we could imagine as design patterns in the style of [10] for parallel or distributed applications. The distributed QuickSort implementation discussed here might be thought of as an instance of an extremely simple pattern of binary distribution. [13] discusses a framework for these sorts of patterns and classifies several.

7.1.3 Designing run-time distribution policies

Finding good heuristics to trigger `split` and `collect` requests for various kinds of parallel and distributed architectures would also be a useful endeavour. Many of the same heuristics that are used currently for process scheduling, such as identifying I/O bound and CPU bound processes, could easily be adapted to the SDC model. In fact, a simple version of this approach was used in a distribution policy tested on Mimoid. The implementation used counters in the `BasicProcessor` objects to determine how many ticks each computation was allocating to memory references and how many it was allocating to receiving and sending messages. Because the Mimoid processor model treated each memory reference and message send or receive as a single tick, this mechanism was somewhat crude, but provided the necessary infrastructure to identify computations that were starved for data or starved for computational power.

In addition, we've spoken rather blithely about the "run-time system" as if it were a single entity that is omniscient and omnipresent. While this assumption holds true for tightly-coupled systems (usually ones in a single box), it is rarely true of distributed systems, and certainly not true of Internet-spanning systems. Thus it becomes pertinent to ask how responsibility for issuing `split` and `collect` requests is distributed. What are the consequences of giving individual processing nodes the ability to trigger "local" distributions and merges, versus giving a central authority with more global information this ability? Similarly, it could be prudent to give applications the ability to hint to the run-time system when it might be appropriate for some computation to `split` or `collect`, similar to how Java, for instance, gives the application the ability to suggest appropriate times for garbage collection to occur[8].

7.1.4 Adapting to a changing environment

Aside from optimizing heuristics, "real world" policy constraints and how to best implement them in an SDC model is an interesting problem. How to fairly allocate processors amongst multiple applications is one such constraint issue. Should the run-time system evenly cut a virtual representation into pieces to present to each

application? If the applications have vastly different computational requirements, this approach could be extremely suboptimal. Factoring in differing application priorities makes this issue even more complex.

Another set of constraints are physical: dealing with the addition and removal of processing nodes and interconnect, either through operator intervention or changes in the functional status of components. While the dynamism of the SDC model makes these constraints easier to deal with–collect away from faulty hardware, split onto new hardware–implementation issues surround how best to allow the application to adjust to these changes. An implementation like Mimoid that requires a computation to immediately respond to `split` and `collect` requests would deal well with these issues. It would be useful to know how a more relaxed implementation could accomodate them.

7.2 Potential applications

There are several classes of application which could benefit from the SDC model. Server applications such as web or database servers are often multi-threaded to take advantage of multiple processors and the asynchronicity of request/response processing. Depending on the rate of network traffic as well as the number of processors and their interconnect, a server application might take different approaches to distributing the workload. The latest version of the Apache web server provides the application programmer with several processing models, each suited to a particular class of architecture. Generalizing this approach could certainly help server applications obtain closer to optimal performance on a range of architectures without requiring platform-specific rewrites.

The emerging field of peer-to-peer applications could also benefit from an SDC approach. On an network as heterogenous as the Internet, it is impossible to rely on the performance of a random network connection or processing node. It is critical that an application adapt to the resources made available to it, and it is hard to imagine that any run-time system, no matter how cleverly written, could intelligently

distribute an application without the kind of feedback available in the SDC model.

Chapter 8

Conclusions

Decisions are made best by those with the best information. For many of the decisions involved in distributing computational tasks, the programmer knows what her application needs to do, and how it is going to go about doing it. If the application can make decisions paramaterized by information about the underlying architecture, it is reasonable to assume that those decisions will be correct.

Correspondingly, only the machine on which the application is running is in a position to know when the assumptions on which those decisions are based–network performance, CPU load, application priority–no longer hold true.

By setting up a feedback loop between the system and the application, such that they jointly make task distribution decisions, the SDC model allows the desiginer of each to specify behavior free of guesswork.

Implementation experience with Mimoid has demonstrated that the asynchronicity forced on the programmer by the SDC model is not a severe burden, as the programmer can implement an entire range of behaviors, paying only for complexity that will allow the application to better adapt for increased performance. In addition, the asynchronous approach encourages a cleaner separation between distribution policy and the underyling algorithm.

It is impossible to abstract the performance characteristics of distributed and parallel architectures: if performance did not matter to, a sequential architecture would be cheaper and easier to program. The lesson of SDC is that an abstraction can

make performace characteristics explicit, without sacrificing efficient resource use or programmability. This lesson is particularly relevant to the software engineering discipline, which has been slow to adapt to novel parallel and distributed architectures.

Bibliography

[1] Google press reviewer's guide.

[2] IBM UNIX servers: ASCI white.

[3] David L. Andrews and Eric Barszcz. *The C* Parallel Programming Language*, volume 6 of *Special Topics in Supercomputing*, chapter 3, pages 93–101. Elsevier Science Publishers B. V., 1992.

[4] Anthony Cataldo. Intel muted ambitious Pentium 4 design. *EE Times*, December 2000.

[5] Peter Coffee. Future chips: Some simple, some sexy. *PC Week*, October 1998.

[6] Kenneth W. Dritz. *Ada Solutions to the Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, chapter 2, pages 10–17. Elsevier Science Publishers B. V., 1992.

[7] Allan Knies et al. IA-64 architecture and compiler technology (ISCA 2000 tutorial). Technical report, Intel Corporation, 2000.

[8] Bill Joy et al. *The Java Language Specification.* Addison-Wesley, second edition, 2000.

[9] Engler et al. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[10] Erich Gamma et al. *Design Patterns.* Addison-Wesley, 1995.

[11] F. Allen et al. Blue gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2), 2001.

[12] Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Finland, June 1997. Springer-Verlag LNCS.

[13] Mani Chandy et al. Research focus: Parallel paradigm integration. *Parallel Computing Research*, July 1993.

[14] Simon Peyton Jones et al eds. Haskell 98: A non-strict, purely functional language. Technical report, haskell.org, February 1999.

[15] Martin Gardner. Mathematical games. *Scientific American*, (223):120–123, October 1970.

[16] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[17] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1989.

[18] C. A. R. Hoare. Algorithm 63, Partition; Algorithm 64, Quicksort. *Communications of the ACM*, 4:321, July 1961.

[19] Rishiyur S. Nikhil and Arvind. *Id: A Language with Implicit Parallelism*, volume 6 of *Special Topics in Supercomputing*, chapter 5, page 170. Elsevier Science Publishers B. V., 1992.

[20] et al Robert Alverson. The Tera computer system. In *ACM International Conference on Supercomputing*, pages 1–6, June 1990.

[21] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.1 Reference Manual*, June 2000.