Applications of Genetic Programming to Parallel System Optimization

by

Robert William Pinder

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author . . . . . .
Department of Electrical Engineering and Computer Science
December 15, 1999

Certified by . . . . . .
Martin C. Rinard
Assistant Professor
Thesis Supervisor

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

Applications of Genetic Programming to Parallel System Optimization

by

Robert William Pinder

Submitted to the Department of Electrical Engineering and Computer Science
on December 15, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

The parallel computers of the future will be both more complex and more varied than the machines of today. With complicated memory hierarchies and layers of parallelism, the task of efficiently distributing data and computation across a parallel system is becoming difficult both for humans and for compilers. Since parallel computers are available in many different configurations, from networked workstations to shared memory machines, porting to new parallel systems is also becoming more challenging. In order to address these problems, this research seeks to develop a method of automatically converting generic parallel code to efficient, highly optimized, machine-specific code.

The approach is to use genetic programming to evolve from the initial parallel program an augmented, performance efficient version of the program specific to the target system. Specifically, the evolutionary system will evolve for every loop a distribution algorithm and a configuration for the work distribution. These algorithms map each iteration of the loop to a specific thread assignment. The algorithms are then inserted into the code along with the necessary parallel constructs. The overall work distribution of which processors are available to each loop is also evolved in order to minimize communication costs and delays. Results show a factor of two speed increase on some tests, and in nearly all cases the system evolved solutions with measurable performance improvements over the best known hand-coded methods. Also, the evolutionary system often derived solutions that solve the scheduling problem in an unintuitive way. The evolutionary system proved to be an effective technique for optimizing work scheduling in parallel systems.

Thesis Supervisor: Martin C. Rinard
Title: Assistant Professor

Acknowledgments

This research would not have been possible without the help of many others. First and foremost, I would like to thank Greg Gaertner of Compaq Computer Corporation. Without his mentorship, advice, and assistance none of this work would have been possible. Also, I would like to thank my manager Greg Tarsa also of Compaq for providing the necessary resources. I am also indebted to Jim Moore of Compaq, whose advice and intuition on genetic algorithms helped me avoid many possible pitfalls. While I can not list all of my Compaq coworkers, I would like to thank all of them for their selfless assistance whenever I needed help. I would also like to thank Professor Martin Rinard of MIT for serving as my academic advisor.

No project that I ever complete in my lifetime could have been done without the loving support of my parents and siblings. This thesis document is no exception. I would also like to thank Yee Lam, Phil Sarin, and Christian Carrillo, for their encouragement, goading, and thoughtful advice, respectively. Finally, I would like to thank Phil Ezolt, Jeremy Lueck, and Chris Rohrs for their help and making my stay at Compaq an enjoyable one.

The software for this work used the GAlib genetic algorithm package, written by Matthew Wall at the Massachusetts Institute of Technology.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Problem with Parallelization

Optimizing the performance of parallel computers is substantially more difficult than optimizing the performance of serial computers. Parallel machines are more complex in implementation, have performance that is difficult to measure, and are widely varied. While it is obvious that parallel machines have more complexity owing to their multiple processors, the supporting components of communication systems and memory hierarchies play a larger role. Parallel machines require some sort of medium for communication, which ranges from low-latency memory sharing to high-latency ethernet. These communication channels have performance characteristics that are difficult for the programmer to estimate, including variable latency and limited bandwidth. Also, no performance analysis is complete without an investigation of the memory system. Locality of data is more important in parallel systems since non-local data must be requested through the communication system. [19] Many processors may share the same low bandwidth interface to the memory, making local caching an even greater performance advantage. In addition to the complexity of parallel systems, optimizing performance is further complicated by measurement difficulties. Parallel programs are often non-deterministic in their running times, making fine tuning impossible. Specialized structures in the operating system or even on-chip counters are required to properly measure parallel performance, so few accurate tools exist. Also,

parallel computers have the extra dimension of space in addition to time—both the processor assignment and the order of execution are important, further taxing our ability to understand the system. Finally, parallel machines are available in many different configurations, ranging from shared memory multiprocessor supercomputers to networked workstation farms. Each has very different performance needs. Consequently, tuning one piece of software for a specific parallel machine will not be helpful when running the software on a different parallel machine. Every piece of software must be tuned for every machine. Optimizing parallel machines is difficult because the performance characteristics are poorly understood and the performance gains are difficult to migrate to different machines.

Despite these difficulties, the traditional approach to performance engineering for parallel machines is to gather information about the performance of the system and then use that knowledge to improve the performance. This technique is used repeatedly for each piece of software and target machine and is usually the work of a team of engineers. Recently, more advanced parallelizing compilers [32] build into the compiler knowledge about many different execution environments and target architectures, and use this knowledge to develop code for a variety of target machines. Both face the previously discussed problems of insufficient knowledge due to complexity and uncertainty in measurement. This problem is not likely to resolve itself, since parallel machines of the future will be even more difficult to optimize. As memory hierarchies become deeper, microprocessors begin to support on-chip multi-threaded parallelism, and communication topologies become more advanced, engineers and compilers will have to constantly update their methods to account for these and many other advances. A new technique that is both automatic and efficient is needed to match the growing complexities of parallel machines.

No matter what sort of performance engineering approach is used, it is essential to improve the three primary performance problems of parallel programs: uneven work distribution, communication costs, and memory access costs. All three of these performance costs may be reduced by improved work scheduling. Obviously, an uneven work distribution is inherently improved by better work scheduling. By placing more

coarse-grain tasks on topologies that have higher communication costs, and more fine-grain tasks on networks with lower communication costs, work distribution can lower the total amount of time the system spends waiting for communication. Also memory access costs can be lowered as the improved work distribution assigns the tasks in a way that improves data locality—essential for effective cache performance. Each of these parallel performance problems is improved by optimizing the work scheduling algorithm.

1.2 An Evolutionary Proposal

The objective is to use genetic programming to automatically evolve nearly optimal parallel code for a specific target machine. Given a piece of generic parallel code, the system will evolve an efficient scheduling and distribution of the program's parallel work. The details are discussed later in this document, but briefly, the system takes as input a program with sections and loops to run in parallel marked with directives and a description of the number of nodes and the number of processors per node in the parallel machine. From just the code and minimal system information, a population of scheduling algorithms and processor distributions are randomly generated for each loop. The performance of each of these potential algorithms is evaluated by running the program with the distribution instrumented into the code. At the end of each generation, the different individuals in the population mate, with the faster algorithms having a higher probability of mating. The new population is tested, and as generations pass the population improves toward nearly optimal performance. [16] Eventually the population converges, and the best algorithms are inserted in the final version of the code. The result is a highly optimized parallel program.

All issues of parallel optimization are not addressed, rather only iteration scheduling and work distribution are considered. Also, serial code will not be converted to parallel. Instead, the focus is scheduling work that is known to be safe to execute in parallel. Other tasks such as identifying potential parallel regions, code restructuring, resolving data dependencies, and data distribution are not supported. These

problems have been addressed by a host of different techniques with some success, but their further investigation is out of the scope of this research. However, given the success of evolving efficient schedulers and work distributions, these other fields may also merit application of genetic programming. More importantly, the two primary causes of parallel performance degradation, uneven work distribution and poor data locality [20] are addressed. By scheduling iterations, the problem of uneven work distribution is explicitly solved, but more over, by reorganizing the iterations the cache misses are reduced and network communication will be lowered by placing processes closer to the data they need. Similarly, problems of inefficient memory access can also be implicitly improved. While not all aspects of parallel optimization are addressed by this research, improving work scheduling does capture a wide range of parallel performance issues.

Evolutionary systems have several advantages. This approach is automatic just like other parallelization schemes, but does not require the performance of the target machine to be well understood. All that is required is the system topology (the number of nodes and processors per node) and what communication protocols are required to communicate between the nodes. Another advantage of evolutionary techniques is that they are likely to garner more impressive performance gains, since it is possible that the system will evolve a work distribution that takes advantage of some poorly understood or perhaps unknown performance characteristics of the target machine. Both fine-grained and coarse-grained parallelism can be exploited by distributing the parallel work in different ways. Finally, the process of evolving many different scheduling algorithms will hopefully yield some insight into the nature of efficient schedulers, and could be used to build better schedulers for machine specific compilers. Each of these possible advantages will be explored.

1.3 Related Work

This research is not the first application of genetic programming to automatic parallelization techniques. The Paragen system [35] employed genetic programming to find

data dependencies and identify parallel regions. Since the process of identifying parallelism has been attempted by using several different techniques, this research only focuses on the scheduling of those parallel regions once they have been identified. The scope of this research is limited to loop scheduling, but it could be easily applied to other parallelization tasks. Other research has applied simulated annealing [36] and other search techniques to the task of optimal work scheduling, but such work has been limited to only a specific parallel architecture, not any general parallel machine.

Current approaches to automatic parallelization do not efficiently optimize for specific target machines. Automatic parallelization research generally applies data dependency analysis to discover the parallelism implicit in the serially written code [28, 19, 10]. The parallelism is then implemented for the target architecture by a compiler into an executable. In order for the compiler to make a truly efficient executable, the performance characteristics of the machine must be well known. More advanced systems [32] build into the compiler knowledge about many different execution environments and target architecture, and use this knowledge to develop code for a variety of target machines. As new technology advances, more work must be expended to keep such compilers current, and as parallel machines become more complex, performance gains will become more difficult to obtain. By not requiring any knowledge about the target system, the evolutionary system can automatically find optimal results for arbitrarily complex systems.

1.4 Document Overview

The rest of this document serves to further explain the use of genetic programming for parallel system optimization, and provide some experiments and benchmarks that demonstrate the success of this technique. Chapter 2 discusses the methods for parsing the input code and generating output code, and Chapter 3 discusses the details of the evolutionary system. Chapter 4 presents a series of experiments and their results. Chapter 5 analyzes the results from those experiments and offers some conclusions.

Chapter 2

Code Generation

2.1 Overview

The process of converting a generic parallel program into a parallel program with efficient work distribution algorithms for a specific target architecture is divided into six phases. First an abstract parallel representation (APR) is created from the serial and parallel loop structures in the original code. The APR is a semantic tree that describes the parallel and serial aspects of the original program. Second, the target architecture description (TAD) is combined with the APR to define the space of possible distribution algorithms. The TAD is a minimal description of the target environment—the performance characteristics of the target system are assumed to be unknown. Then the different semantic structures are broken into loop entities, which are sections of the program which are independent and can be evolved separately. Next, a population of possible distribution algorithms are randomly generated for every loop, along with a population of thread teams for every loop entity. These populations evolve through natural selection and crossover for a fixed number of generations or until the population has sufficiently converged. Finally, the different distribution algorithms are analyzed, and the most fit are inserted into the final code. Each of these steps and the associated structures are discussed in the subsequent sections.

2.2 Generic Parallel Programs

A generic parallel program is a program with a non-implementation specific encoding of the divisions of parallel work. Currently there are many ways to implement a parallel algorithm, ranging from implicitly parallel languages such as Parallel Haskell [3] to explicitly parallel communication protocols such as Message Passing Interface (MPI). Each of these implementation techniques has its own advantages and disadvantages. One goal is to use an evolutionary system to leverage the advantages of certain protocols dependent on the specific target architecture. A generic parallel program then should be independent of both the implementation of the parallelism and the target machine. Since current parallel techniques do not provide both of these elements, a different abstract parallel representation is needed. The APR is derived from directives that mark parallel regions in the source code, and little else; most of details of the parallel implementation are left to the evolutionary system. Specifically, the directives mark the loops that should be parallelized, but the details of protecting and updating shared variables, distributing private variables, and relaxation techniques are left for the evolutionary system to evolve an efficient technique based on the target architecture. The details of the APR directives are discussed in the next section, but first it is necessary to further examine the importance of a directive based parallel representation.

The directive based approach is both easy to use and provides a clear parallel framework. The programmer does not need to implement a complicated communication scheme for every potential target architecture. Also, directive based parallel programming has a long tradition in the parallel programming community, with the most recent incarnation being the OpenMP standard. [29] Serial programs are easily converted to parallel by simply inserting parallelization directives. Parallel programmers are accustomed to such techniques. Given such industry familiarity, using a previously defined directive set was considered, but such directives tend to be both overly complicated and not expressive enough to handle the general case. While directives do limit the types of parallelism the programmer can use, most of the implementation

of the parallelism is decided by the evolutionary system, so such limitations are not a concern. More important than the advantages of ease of use, the directive based approach allows the code to be divided into easily configurable regions. By identifying the levels of nested parallelism and parallelizable loop iterations, a framework for all the potential distributions is revealed, and hence the evolutionary space is defined. Thus, directive based parallelism provides a clear mechanism to identify parallel work.

2.3 The Components

2.3.1 Abstract Parallel Representation

The abstract parallel representation is a description of the structure of the parallel code. The APR is implemented as a list, with the first node representing the first serial section of the code, and each subsequent node representing the serial and parallel sections of the code at the top level. Loops within a parallel section are represented as children of the parallel section that they are nested within. The children are ordered, so the left most child appears first in the original program and each additional child to the right is associated with the next block of source code. Nested loops are represented as children of the loops they are lexically nested within.

Each node is not just used to represent the structure of the original program. The parallel loop nodes store a pointer to the data distribution algorithm used for the loop, the variables used to store the maximum number of loop iterations, the type of communication system that is used to distribute the loop iterations, and the nodes in the target architecture where work will be sent. The serial nodes do not store any additional information. While the structure of the APR is not changed, the parameters in the parallel loop nodes are evolved. Each loop has specific work distribution needs, so each loop has a different distribution algorithm. Also, this approach allows nested loops to distribute the data in a layered fashion, with the outer loops dividing the work to sections of the target architecture, and the inner loops dividing the work within those sections.

serial Marks code to be run in serial. No parallelization is attempted in this region.

parallel(*variable list*) Marks the region to be run in parallel. Those variables which must be private or shared among all processes are suitably denoted in the *variable list*.

pdo(*variable list*) Marks a loop to be included in the evolutionary process. The *variable list* consists of the program variables that define the maximum iteration, the current iteration, and a user defined parameter for that loop.

pend Marks the end of a loop to be optimized.

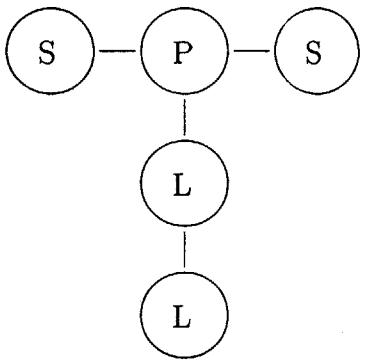

Figure 2-1: Serial, Loop, and Parallel Nodes in an example program

A parser is used to build the APR from the program code. Directives mark the separate regions of code, and the parser uses these directives to identify the code to be run in parallel. A more sophisticated system could automatically identify the parallel regions, but that is beyond the scope of this research. Each of the directives is described below.

Figure 2-1 and figure 2-2 provide an example of sample code and the resulting APR structure.

2.3.2 Target Architecture Description

The target architecture description is an encoding of the parallel system, including the number of processors and the communication protocol the processors use (network or shared memory). In the case of clusters of shared memory machines, some processors

```
program test
...
!$APR parallel(i,j)

!$APR pdo(20,i,1)
  do i=1,20
!$APR pdo(i,j,1)
    do j=1,i
      a(i) = i+j
    end do
!$APR pend
  end do
!$APR pend
!$APR serial
  print *, a(20)
...
end program
```

Figure 2-2: Fortran code with directives to generate example APR

are capable of communicating to others in different ways. This information is not used to guide the evolutionary process, but instead it is used to generate the correct code so the evolved algorithms are implemented correctly for the target machine. The TAD and APR are used to define the space of possible distribution algorithms. The TAD defines the number of processors and the APR defines the variable names for input to the algorithm.

2.3.3 Distribution Algorithms

The distribution algorithms are of central importance. Essentially, the distribution algorithms map the iteration space of a loop to the processor space of the target machine. The algorithms are functions of three variables, the current iteration, the maximum number of iterations, and a user defined parameter. For output they return a thread assignment for that iteration.

The format of a distribution algorithm is an expression tree with two types of nodes. The leaf nodes are terminals, consisting of the first ten prime numbers and

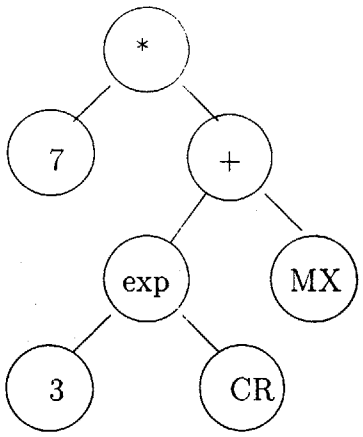

Figure 2-3: An Expression Tree: CR refers to the current iteration, and MX the maximum number of iterations.

the three input variables. The branch nodes are operators, including the simple arithmetic functions, $+$, $-$, $*$, zero-safe-divide, trigonometric functions sine, and an exponentiation operator. All the operators are protected from overflow or other run-time errors. To execute the algorithm, the variables are set, and then the expressions are evaluated from the bottom of the tree up. The expressions do not allow changes of state or recursion, so the execution time is always finite. The result of the algorithm is modular divided by the number of threads, and the result is the thread assignment for that iteration. The modular division ensures that a valid thread assignment is always generated.

The expression tree in figure 2-3 refers to the equation:

$$ThreadAssignment = (7 * (e^{3*CR} + MX))$$

where CR refers to the current iteration, and MX refers to the maximum iteration.

2.3.4 Loop Entities

A loop entity is a set of loops that are only dependent on the other loops in the set. Any loop which is lexically nested within another loop is dependent on that loop, and any loop which has loops nested within it is dependent on those loops. Loops which

are not lexically nested are in separate loop entities. Since the performance of the loops in a loop entity depends only on the other loops in the set, each loop entity is evolved separately. The evolutionary parameters of a loop entity are the processor distribution suggested by the thread team and the distribution algorithm assigned to each loop in the loop entity.

2.3.5 Thread Teams

A thread team is a structure for dividing the available processors among the loops in a loop entity. The thread team describes which processors the loop can run on, and then the distribution algorithm assigned to the loop distributes the iterations among those processors.

The idea of thread teams is important, because it provides support for layered parallelism. In cluster configurations, the associated software supports layers of parallelism, with each layer having varying communication costs. For a cluster of shared memory machines, to two layers are intra-node communication and inter-node communication. With non-uniform access memory and on-chip multi-threaded designs, many more layers are possible. Optimizing the granularity of the computation with the appropriate level is crucial to performance. Often it is appropriate to do more coarse-grain computation among processors with high communication latencies, and to distribute the more fine-grain work among processors that can quickly communicate. Also, some target systems may have processors that can not directly communicate, so having a team leader handle the communication is necessary. Thread teams allow a logical division of the processor space that supports layered parallelism found in complex hardware configurations.

The thread teams are a tree structure derived from the APR. They have the same structure as the semantic structure of the lexically nested loops in the original program. The base node is the outer most loop, and each child of the node is a nested loop within the upper node. Each node of the tree is assigned a number representing the number of threads available for the loop to distribute iterations over. Tracing down the nodes of the tree determines which processors are available for each loop

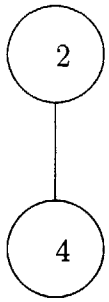

Figure 2-4: A Thread Team for a four processor system

in the original program. Based on these available processors, the system generates code that distributes to threads on the machine where the processors are located. Each node in the tree is a factor of the total number of processors available. The loop may distribute work to that quantity of processors, but the physical location of the processors is fixed by the target architecture. The code that is generated will favor evenly distributing the available processors across the different nodes of a system, rather than on the same machine. Loops lexically deeper in the tree have access to all the processors of the loops they are lexically nested within, but may also be able to distribute to additional processors. To ensure that every thread team covers all possible processors, the leaf nodes (the deepest lexically nested loops) can distribute to all the processors in the system. Hence, in every thread team tree, the root node has the fewest number of processors, and the leaves can distribute to all of the processors in the system.

The example code in figure 2-2 corresponds to the thread team in figure 2-4 and a target architecture that has four total processors. The shown configuration, 2-4, allows the outer loop to distribute iterations to two threads, and the inner loop to all four available threads. The other possible configurations are 1-4, where the outer loop is serial, or 4-4, where the outer loop is distributed parallel across all processors.

It is important to note that just because a thread team allows a loop to distribute to all the processors, does not mean that the loop must distribute to all the processors.

The distribution algorithm for the loop may choose to only distribute to one processor. The thread teams vary the granularity of the parallelism, by giving the evolutionary system a way of evolving the communication patters of the loops. By only exposing certain processors to loops that may have coarse-grain work, they may find more optimal solutions than if distributing on all the processors. The distribution algorithm manages the work load balancing, but the thread team defines the size of the loads.

Figure 2-5 shows the relationship between the APR, the thread teams, and the distribution algorithms. Each parallel node in the APR (denoted with a "P") marks a separate loop entity. Each loop entity is associated with a thread team that has the same structure as the loop structure in the initial program. Each loop is associated with a separate distribution algorithm, and each distribution algorithm contains an expression tree.

2.3.6 Implementing Parallelism

The actual implementation of the parallelism is flexible. The GA can utilize multiple parallel protocols simultaneously to maximize performance dependent on the requirements of a specific piece of code and target architecture. The implementation protocols chosen for this system are described in this section.

Two parallel communication protocols are used to implement the parallelism in the generated code: OpenMP [29] and the Message Passing Interface (MPI) [25]. Each has very different performance advantages and restrictions. OpenMP is a directive based technique, much like the directives presented in this research. Parallel regions, critical sections, and loop scheduling techniques are all explicitly implemented by annotations in the code. When the source is compiled, the directives cause extra libraries to be included and add code to distribute the work, protect the variables, and manage the locks. OpenMP assumes that all processors are capable of accessing the same memory, so it only works on shared memory machines, not cluster configurations. MPI is different in that it consists of a library of functions that allow the programmer to directly implement parallelism. It provides an interface for each of the processes to send and receive messages. More complicated communication involving broadcast

and communication groups is also available. MPI programs have separate memory spaces, so they are usually used for network systems, but can also be used on shared memory systems. [26]

Both communication protocols are used. First, in order to support communication on both cluster and shared memory systems, but more importantly because they have different performance characteristics. Communication for OpenMP is a simple variable read from memory, yet in MPI a message must be assembled, sent, received, and unpacked. OpenMP is therefore used for fine-grain parallelism, where low latency communication is appropriate, where MPI is used for more coarse-grain tasks. [15] Often times too much fine-grain parallelism can cause bus saturation and slow the overall execution time. [9] In such cases MPI is appropriate. It is often difficult to determine where these divisions should be drawn, so the type of communication protocol is evolved via the thread teams. For a cluster of shared memory machines, the thread team for a loop entity dictates which processors to distribute the work over, and code is generated for either MPI communication, OpenMP, or both, depending on where the processors are located and what sort of communication is needed.

The initial parallel programs are generic in their implementation, so inserting the appropriate OpenMP and MPI code and ensuring correctness requires careful attention to the memory consistency. Since OpenMP supports a shared memory model, the variables are managed by the OpenMP library as specified in the APR directive variable list. The MPI parallelism does not support a global variable space, so the public variables must be explicitly managed with messages. A master node is designated that accepts all the data, combines it appropriately, and redistributes it to the appropriate nodes. Both the master and the slave nodes contain the work distribution algorithms, so all the nodes can calculate the data that must be sent and received. Dependant on the type of parallelism used to distribute the loop, different memory management schemes are used.

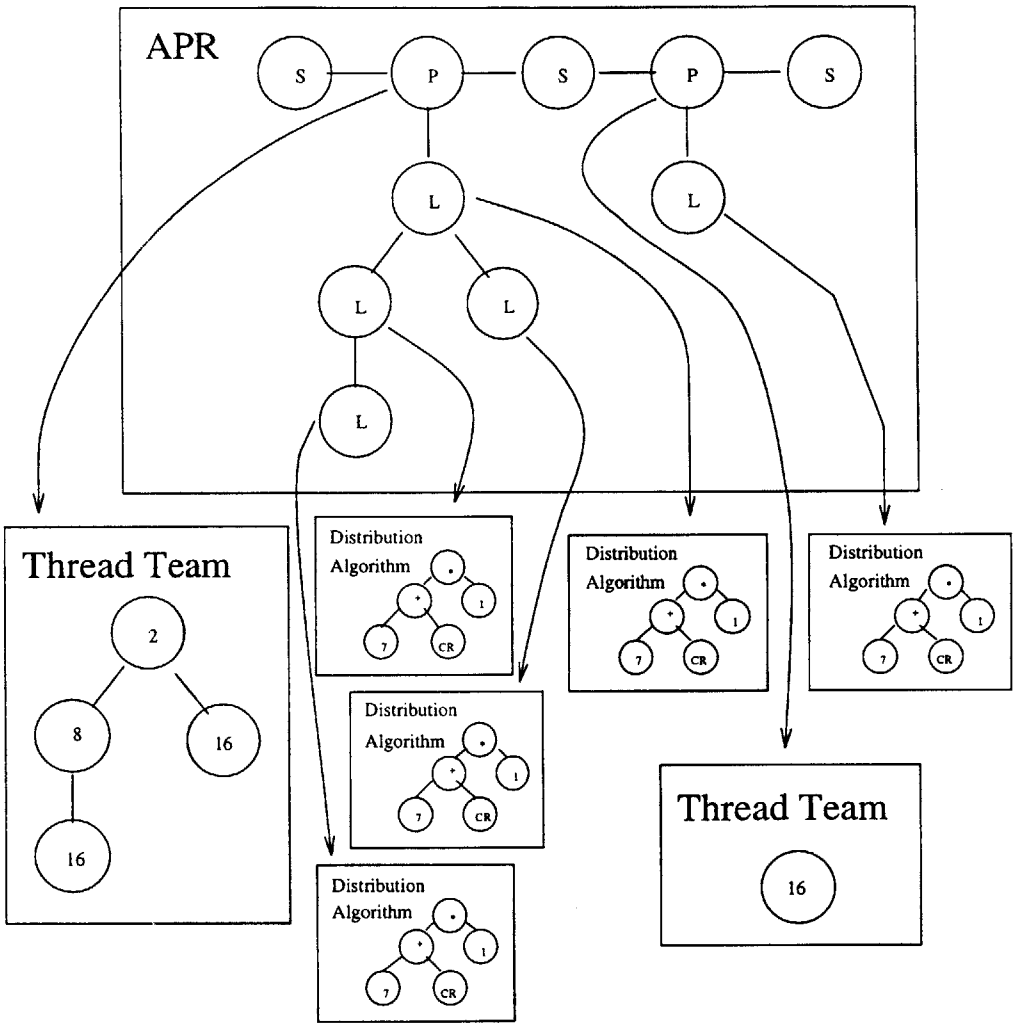

Figure 2-5: A Sample APR and the Associated Thread Teams and Distribution Algorithms

Chapter 3

The Evolutionary System

3.1 Introduction to Genetic Programming

Genetic Programming (GP) is an evolutionary system similar to a genetic algorithm (GA). [22] Essentially, a GA is a search algorithm that is founded on the principles of natural selection. A search space is represented by a genome, an encoding of information that supports the biological operations of mutation and crossover, such as DNA in most living creatures. An individual's genetic material defines that individual's set of traits and hence fitness. More fit individuals correspond to better performance in the original search space. To solve a problem with a genetic algorithm, it is necessary to express the search parameters as a genome and to express the search goals as an objective function used to evaluate an individual's fitness. [11]

The algorithm begins by randomly generating a set of individuals, each having their own genetic material. Each individual's fitness is tested and then ranked by its fitness. The next phase is selection, where individuals are chosen to mate and pass their genetic information to the next generation. The selection is probabilistic, with the more fit individuals having a higher probability of mating. Each offspring inherits some of its traits from each parent, and is subject to possible mutations leading to different traits. After many generations, the population converges toward an optimal position in the search space. [16]

The primary advantage of evolutionary systems over traditional search techniques

is robustness– the ability to rapidly find a solution in an arbitrarily complex search space. [11] While other approaches require a well-behaved, well-understood search space, genetic algorithms only require that the search space can be defined by a genome. While genetic algorithms have probabilistic elements, they are more efficient than random searches because of the incorporation of a fitness function. Also, GA's explore the search space from an entire population of individuals, not just one point. By varying the population size and number of generations, genetic algorithms can perform well for arbitrarily complex search spaces. [17] As much as evolutionary systems are robust, if fitness evaluation is computationally expensive, they are equally time consuming. Genetic algorithms require $p * g$ fitness evaluations, where p is the population size and g is the number of generations. A trade-off exists, greater robustness can be achieved at the expense of more computation.

Genetic Programming differs from traditional genetic algorithms in that the genome is the set of valid computer programs. Actual source code is used as the genetic encoding. GP is a technique for computers to evolve their own programs to solve problems. [22] In the case of this research, the distribution algorithms are sub-programs simultaneously evolved with thread teams in a symbiotic genetic algorithm. The system employs genetic programming to develop each of the individual populations, and then a genetic algorithm to handle the interaction of all the components.

3.2 System Design: The Symbiotic Genetic Algorithm

The evolutionary system is responsible for evolving a distribution algorithm for each loop and a thread team to define the processor space for each loop entity. These different genomes require different genetic operations of crossover, mutation, and random generation. Each of these techniques are described below, in addition to the symbiotic GA which manages all of the different populations.

3.2.1 Genetic Representation of Distribution Algorithms

A distribution algorithm is a sub-program represented as a parse tree for an expression that relates loop iterations to a thread assignment. Since every loop in a program has different performance characteristics, the GA maintains separate populations of distribution algorithms for each loop.

The genetic operations of crossover and mutation are sub-tree manipulations defined for the tree-structures that represent the distribution algorithm. Crossover is implemented by choosing a node in each parent tree, and then swapping the sub-trees. If one of the nodes chosen is the root node of the parent, then the entire tree will be a subtree in one of the offspring, and the other offspring will be a subtree from the other parent. Any node in the tree can be swapped, but the upper nodes are favored more than the leaf nodes. Since the trees are binary, the leaf nodes consist of approximately half of the total nodes in the tree. If all the nodes are chosen with equal probability, then 75% of the crossovers will involve swapping a leaf node from one of the parents. However, favoring swapping the upper nodes leads to better convergence and efficiency. [22] Swapping the upper nodes encourages the formation of distinct subtrees that handle individual tasks, much like subroutines. Also, swapping the leaf nodes often has little effect on the overall program, while swapping subcomponents more quickly combines the traits of the two parents. The probability of choosing a non-leaf node for crossover is one of the configurable parameters for the GA.

Mutation is supported in two different ways, node alteration and node deletion. Node alteration chooses a node in the tree and replaces it with a randomly chosen node. For example, a leaf node may be replaced with a different constant or variable, and a non-leaf node would be replaced with a different operator. Node deletion is intuitive—a node and its subtree are removed from the tree. These two operations help maintain diversity in the genome in order to counterbalance the selective pressure.

A key property of these genetic operations on tree structures is that every possible result from crossover or mutation is a valid distribution algorithm. It will neither

perform an undefined operation nor loop indefinitely on any input. This closure property would not be the case on an arbitrary piece of source code represented as text. Converting the source code into a parse tree is imperative. [23]

The random population generator creates the initial populations. The expressions are created by starting with the root operator node and randomly choosing the arguments, with each possible entry having a fixed probability. If an operator is chosen for the argument, then a sub-expression is created, and the arguments for that sub-expression are also randomly chosen. If a terminal is chosen, then that part of the tree stops growing. The best set of relative probabilities for the different elements in the expression is not known and is part of this research. As the tree increases in depth, the probability of choosing a terminal increases in order to ensure expressions with finite length. In this manner, a population of random expressions is created for every loop in the program.

3.2.2 Genetic Representation of Thread Teams

A thread team is implemented a tree structure that describes the available processors for every loop in a loop entity. Each node identifies the number of threads available to that loop. For each loop entity in the program, the GA maintains a separate population of thread teams.

Like the distribution algorithms, the thread teams have a tree structure. Since the structure of the tree is fixed, the thread teams can not support the same type of crossover and mutation operations. Crossover is the same subtree exchange, but the two parents must exchange the same nodes, or the structure of the offspring would not match the structure of the loop entity and the original program. Also, if the crossover results in an invalid thread team, such as a lower node not being able to distribute work to the same number of processors as a higher node, then the team must be checked and fixed. Because of this potential, mutation is not supported. Since the structures are relatively small, mutation is not a useful operation given the high probability of generating invalid thread teams. [24]

The structure of the thread team is fixed by the original program's loop structure,

but the different parameters at the nodes are randomly generated. The upper nodes are generated first, and then the lower nodes are defined out of the possible teams that could exist given the upper nodes. The leaf nodes are all set to the maximum number of processors.

3.2.3 The Symbiotic Genetic Algorithm

The symbiotic genetic algorithm is responsible for managing the evolution of the separate thread team and distribution algorithm populations by using shared fitness, multiple testing, and migration. Individuals from each of the disparate populations are assembled together to form a parallel program, and then the code is compiled, executed, and timed, and the fitness values are reported back to each of the populations. The evaluation stage has some complications owing to the nature of the problem. First, each of the loop entities is separately timed by inserted timing code, but a loop entity consists of several evolved distribution algorithms and a thread team. Each of the distribution algorithms in the run and the thread team receive the same fitness score, though they have very different fitness characteristics. Since the performance of all of the loops in a loop entity are dependent, they receive the same score. It may be the case that assigning all the distribution algorithms in a loop the same score distorts the actual fitness, but it is difficult to know what the fitness is of an inner loop independent of the outer loop. The key goal is to minimize the overall running time of the loop entity, not the individual loops, so whatever assortment of distribution algorithms that causes the running time to minimized will be encouraged by the selection pressure. Unlike the distribution algorithms, the thread team should have an independent score and be tested over several different distribution algorithms to determine its fitness score. Since the number of possible thread teams is relatively small, the thread team populations are smaller than the distribution algorithm population. Consequently, each thread team individual is tested in several parallel programs. The fitness of a thread team is the average of all the different execution times. In this way the thread teams and the distribution algorithms complement each other. The inadequacies in one can be evolved in the other

to still achieve near optimal performance. Finally, individuals from one population can migrate into populations of the same type of individuals. While each loop will require its own specialized distribution algorithm, especially good individuals from one population should be helpful for the evolution of a different population. The symbiotic GA employs shared fitness, multiple testing, and migration to maximize the potential for optimal convergence for parallel programs.

The evolutionary system described here further extends the biological metaphor of genetic algorithms. The symbiotic genetic algorithm is so named because it implements the population genetics principles of speciation, inter-population migration, and coevolution. The evolutionary environment contains multiple populations of different types of individuals with different genomes. In this sense, the different populations are like an ecosystem, with many distribution algorithm populations and a few thread team populations. Each separate population evolves differently dependent on the specific performance characteristics of the individual loop. Despite having a common random initialization, the different loop populations quickly develop into different "subspecies." Every generation some individuals migrate into the neighboring population. Migration improves convergence since there are likely to be elements of distribution algorithms that are universally good. [12] Also migration should help maintain genetic diversity and prevent a population from becoming stuck in a suboptimal niche by providing a constant influx of different genes. [18] In addition to the interactions of separate loop populations, the thread team populations also interact with the loop populations. The performance of the thread teams directly depends on the distribution algorithms in that loop entity. The success of both types of entities depends on mutually beneficial traits. This symbiosis should help improve performance as the separate populations co-evolve to overcome the disabilities of the other. By utilizing principles from biological ecosystems, the symbiotic GA should find efficient solutions.

3.3 The Search Space

The nature of the search space strongly effects the efficiency of the genetic algorithm. A gradually sloped space with single prominent global maximum is easily scaled and surmounted by the GA, however a search space with many low local maxima and only a single, obscure global maximum is much more difficult. The local maxima often serve as evolutionary "dead ends" as the algorithm prematurely converges on a sub-optimal result. Any problem would be easily solved with a large population for many generations, but that is rarely computationally feasible. [22] A single, prominent global maximum is best approached with a smaller population for more generations with a low mutation rate. More generations give the algorithm time to find the very top of the maximum. A more hilly search space requires a larger population for a fewer number of generations and a higher mutation rate. [11] This more random search helps the GA locate the highest peak among many. Of course, the search space is n-dimensional, so it is not simply "hilly" or "flat," but this is for the purpose of visualization these terms are used. Figures 3-1 and 3-2 further illustrate the different search spaces. In order to efficiently evolve solutions, the nature of the search space must be considered when choosing parameters for the GA.

The search space for the problem of loop iteration assignment for a thread is immense, yet it is completely covered by the genome representation. Describing the search space as all mappings from loop iterations to thread assignments, there are t^n members where t is the number of threads and n is the number of loop iterations to be mapped. Despite this large size, the distribution algorithms cover the set completely. This fact may not appear intuitive, and the proof is strait-forward and illuminative, so it is included below. Consider the function

$$a_1(x/1) + a_2(x/2) + a_3(x/3) + a_4(x/4) + ... + a_n(x/n) = t_x$$

where again n is the total number of iterations, x is the current iteration number, a_n is an integer coefficient, and t_x is the thread assignment for iteration x. This function can be expressed by a distribution algorithm, since it contains only the simple

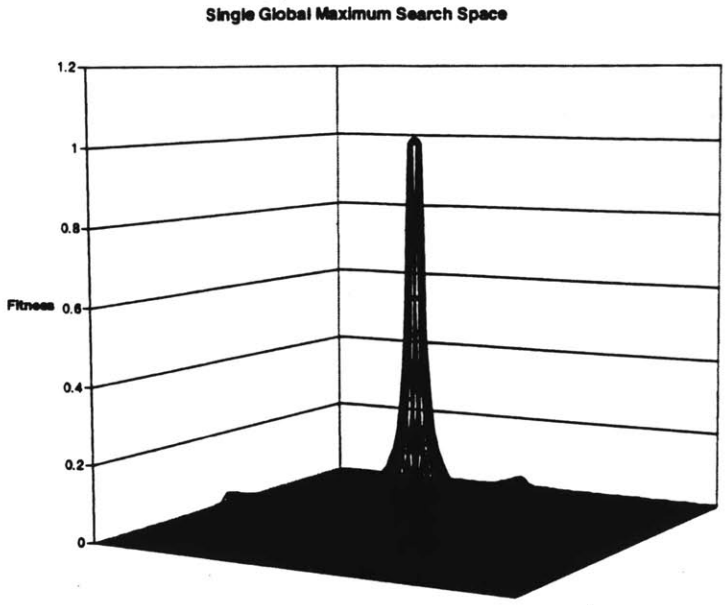

Figure 3-1: A Single Global Maximum

operations of of multiplication, addition, and integer division. This function can also be expressed as the multiplication of two vectors, $\begin{bmatrix} (x/1) & (x/2) & (x/3) & \ldots & (x/n) \end{bmatrix}$ and $\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \ldots \\ a_n \end{bmatrix}$ Now consider a (n x n) matrix, where the ith row is results of evaluating the x vector with $x = i$. Multiplying this resulting matrix with the coefficient vector yields a thread assignments vector, where t_i is the thread assignment when $x = i$.

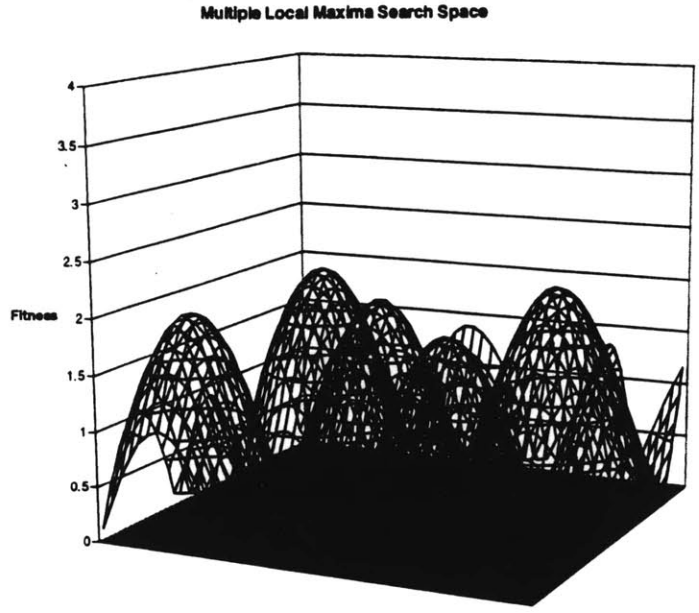

Figure 3-2: A Hilly Search Space

Consider the $n = 5$ case of $Xa = t$ below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 2 & 1 & 1 & 0 \\ 5 & 2 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{bmatrix}$$

This matrix is X has linearly independent columns, since the only solution to $Xa = 0$ is $a = 0$. The matrix then completely spans the 5-space, so for any vector t there exists a vector a that satisfies $Xa = t$. The columns are not just independent in the $n = 5$ case, but are also independent for all settings of n, because the matrix is always lower triangular with 1's on the diagonal. This result is not sufficient since it must also be shown that a is only composed of integers, because the evolutionary system is not capable of evolving arbitrary real numbers.

For this result, a simple proof by induction is necessary. First, assume that at the kth position, all a_1 to a_k are integers. This assertion is valid in the base case, $k = 1$. The first row has 1 in the first position, and zeros in all the others, so $a_1 = t_1$. Therefore, a_1 must be an integer. In the a_{k+1} case, positions 1 through k in the row have integer values (since they are derived from integer division in the original equation), and the $k+1$ position is equal to 1. From the assumption, all a_1 to a_k are integers, so the resulting equation is

$$\begin{bmatrix} x_{k+1,1} & x_{k+1,2} & \cdots & x_{k+1,k} & 1 & 0 & \ldots & 0 \end{bmatrix} * \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_k \\ a_{k+1} \\ \cdots \end{bmatrix} = t_{k+1}$$

$$I_{sum} = a_1 x_{k+1,1} + a_2 x_{k+1,2} + \ldots + a_k x_{k+1,k}$$

$$I_{sum} + a_{k+1} = t_{k+1}$$

This reduces to $a_{k+1} = t_{k+1} - I_{sum}$, and since both t_{k+1} and I_{sum} are integers, a_{k+1} is an integer. Since both the base case and inductive step are true, there exists for every vector t a set of integer coefficients a that leads to that scheduling allocation. To implement any set of thread assignments, it is only necessary to choose the correct set of coefficients.

This function is one of just many functional forms that covers the entire search space. Certainly many other forms exist. Since the end result of the function is modular divided, many coefficient sets yield the same thread distribution. Also, many different thread distributions yield the same performance results. When all the processors are equivalent in a system, a simple renumbering of the threads will cause the same result. Moreover, the execution time of some iterations is smaller than

the timing granularity, so their assignment does not effect the final result. Although the search space is large, the reduction to the performance space is smaller. Any single optimal result will have many functions that are varied both in form and in constants that yield that same optimal result. The search space therefore has many local maximums, some of which are equivalent global maximums.

Given that there exists a single functional form that covers the space, it is not initially clear why it is appropriate to evolve algorithms instead of coefficient sets or mapping arrays. Such coefficient sets are only valid for a set number of iterations. It is often the case that loops are executed multiple times with a variable number of iterations. Consequently, a function which covers all of those possible maximum iterations is also needed—a static coefficient set is not adequate. In addition, it is hoped that the evolved functions will lend insight into the nature of the distribution. Perhaps a set of consistently effective loop distribution algorithms will be discovered. These effective algorithms could be inserted into similar code, and the evolutionary process will be circumvented.

The search space of thread assignments for given loop iterations is very large, yet it is completely covered by the set of all possible distribution algorithms. There exists for every thread assignment vector a distribution algorithm that maps the set of integers to that vector. The search space when expressed as distribution algorithms has many local maximums and several equivalent global maximum. A principle aspect is to find genetic algorithm parameters that perform well in this search environment.

3.4 Algorithm Tuning and Calibration of Parameters

Proper calibration of the evolutionary system is a key part of this research. The evolutionary system has hundreds of different parameters, and finding the proper set of parameters that guarantee a good result is difficult. Every potential program and target architecture have an ideal set of evolutionary parameters that can only

be determined after many trail runs. After some initial tests, a set of evolutionary parameters which work well for many different test cases were found. The most important parameters are described in the following sections.

3.4.1 Standard Crossover Techniques are Used

The two crossover parameters are the crossover rate and the node selection parameter. The crossover rate is the probability that the offspring of two individuals are the result of crossover instead of cloning. These tests use a crossover rate of 1, because the replacement strategy does not remove the parents from the population unless they are less fit than their offspring. The node selection parameter is the probability of selecting a non-leaf node for the crossover point in a tree. A node selection parameter of 0.9 was used, which is widely suggested in the literature. [22] Previous runs with a node selection parameter of 0.5 resulted in slower convergence and lower probability of finding an optimal result.

3.4.2 Mutation is Minimized

The mutation in the system is controlled by three parameters, the mutation rate, the mutation spikes, and the spike period. The mutation rate is the probability that an individual will have a mutation when created. Often times in genetic programming, mutation is not used because lack of diversity in the population is rarely a problem. These tests use a low rate of mutation (0.05). It was rare that the mutation was necessary, but sometimes in small populations certain alleles became extinct, so mutation helped restore those lost genes. Mutation spikes are akin to giant radiation storms or comets striking the Earth. They are intended to perturb a converging population in an attempt to reinsert diversity into the population and possibly find a superior result. The mutation spike is implemented by cloning the entire population under a high mutation rate. The spike period is the number of generations that pass between each spike. Experimentation with mutation spikes in this research found them generally ineffective. At times they even served to decrease diversity by crippling

competing individuals and permitting one individual to rise out of the aftermath. The single surviving individual rapidly reproduces itself, and quickly dominates the population—leading to an overall loss of diversity. Mutation rarely added diversity to the population and also slows convergence, so the system uses a low level of mutation.

3.4.3 Replacement Strategy: Direct Removal or Elitism

The two primary replacement parameters are the number of new individuals in each generation and how to replace the old population. Once the new individuals are created, there are two methods for replacement, direct removal and elitism. Direct removal sorts the old population by fitness and then removes the number of individuals equal to the number of new individuals. Then it inserts the new individuals into the population. Elitism only inserts the new individuals if the old individuals have lower fitness than the new individuals. This research uses elitism, which has been shown to cause faster convergence in populations. The disadvantage of this technique is that it may cause an overly rapid loss of diversity and converge too quickly on an non-optimal result. The advantages of faster convergence outweigh any diversity concerns, so elitism is used throughout the tests.

3.4.4 Multiple Tests Improve Fitness Calculations

Parallel programs are notoriously non-deterministic in their running time and are often difficult to time accurately. Inaccurate fitness scores undermine the evolutionary process and make finding an optimal result less likely. A variety of statistical techniques can be used to address this problem, but one of the simpler methods is to test the program multiple times. Multiple execution of the test program is especially useful in networked systems where communication has highly variable latency. In such cases, the program is executed multiple times and the fitness is the average of those running times.

3.4.5 Criteria for the Objective Function and Selection Techniques

At first glance, the objective function simply records the running time of the program; however, there are several subtleties in both the measurement of the running time and the use of the running time as the objective function. First, there are several timing options. One could sum the user time spent on all processes, record the total time spent on all processes, or simply use the wall clock change. The most important quantity is to minimize the wall clock time, since the individual process times may not be a good reflection of the actual program running time. Accordingly, the objective function seeks to minimize the wall clock time. Much research has been done in ways to relate the raw fitness score to a parameter for selecting the individuals for mating. Often the raw fitness score is scaled by a constant factor or a function to cause differentiation between individuals with nearly equal fitness scores. Also, a variety of schemes are used to select individuals from ranking and assigning a probability, or relating the probability of selection directly to the fitness via a function. Each technique has different advantages dependent on the search space. [5] Given the generally amicable nature of the search space, the raw fitness is used for the selection parameter. The roulette wheel technique [39] is used to select a pair of parents. In this technique, each individual's probability of selection is equal to the percentage of the total fitness score over all individuals. Hence, each individual has a slice of the roulette wheel, directly proportional to their percentage of the total fitness. This selection technique seems to work well, so other techniques are not used.

3.4.6 Effects of Altering the Initial Population

The primary parameter that effects the initial populations for the distribution algorithms is the expected size of the expression tree, which is related to the probability that during construction, a new node is not a terminal node. Initial trial runs revealed that the final populations were always much larger than the initial populations. Also, trails with smaller trees in the final population tended to perform worse than larger

tree results. The parameters were increased to make the average depth of initial population trees equal to four. This change did cause results with superfluous subtrees in the final population, but the results in general caused faster convergence and a much higher likelihood of a near optimal final solution.

3.4.7 Population Size and Generation Number

The population number and generation size are related in that they define the running time of the algorithm and the probability of obtaining an optimal result. [11] A considerable amount of experimentation in this research attempted to find a good setting for these parameters. In general, large populations for fewer generations faired better than small populations for more generations. The results from experiments varying these parameters need further attention and are discussed in section 4.3.3.

Chapter 4

Results

4.1 Overview

This chapter explains the various experiments used to test the effectiveness of the evolutionary system. First, a set of initial experiments are explained that describe the testing environment. The next section discusses the convergence properties of the GA, and the optimal settings of the various parameters dependent on the problem type. The last section describes a set of more comprehensive experiments on a variety of target architectures and programs with varied performance characteristics.

4.2 Results of Initial Trials

The initial tests are simple programs executed on a four processor shared memory machines (SMM). The test programs have one loop with 20 iterations and the iterations have no data dependencies. Three different loop work distributions were tested: uniform, linear, and exponential. The distribution names denote the way the work is distributed per iteration of the loop. The uniform distribution has the same amount of work per iteration, and the linear and exponential distribution have a linearly and exponentially increasing work distribution, respectively. Since the target machine has shared memory, only OpenMP was used to distribute the work.

The evolved results are compared against control programs. These programs are

Static Distribution Assignments are determined at compile time, and this technique gives each thread an equal number of iterations.

Dynamic A run-time technique, the threads poll a central manager thread which distributes to the thread the next iteration that must be computed.

Guided This scheduling is a static technique, but it distributes the initial iterations in larger chunks, lowering the communication costs. This technique is appropriate when the work load is increasing, so the iterations with less work are grouped together.

intended to accurately model the parallel implementation using conventional, hand-coded techniques. The control programs implement the three primary traditional methods for parallel scheduling: static, dynamic, and guided.

Each program was tested with all three of these techniques, and the best result was compared to the best evolved result. Figure 4-1 contains a table with results for each of the three distribution types, with comparison to the other hand-coded techniques. These results are normalized, so a score of 1.00 denotes perfect scaling for four processors, and 4.00 denotes the serial time.

Distribution Type	Uniform	Linear	Exponential
Serial Time	4.00	4.00	4.00
Hand-coded Time	1.00	1.06	1.21
Evolved Time	1.00	1.00	1.01

Figure 4-1: Normalized Performance Times for Three Distributions

In each of these cases, the evolutionary system is able to evolve a solution that was at least as fast as the best hand-coded solution. Also, the solutions are nearly optimal; only in the case of the exponential distribution is the result less than perfect scaling. The more complex the work distribution, the more the GA is able to profit over the hand-coded results. The evolved results for exponential distribution is 20% faster than the hand-coded version.

These results took little computational time. In one hour, the evolutionary system often finds the best result, and all of these experiments ran for less than three hours. The GA finds nearly optimal solutions, and rapidly converges on such results.

These tests are simple, but they are representative of some types of problems the evolutionary system can solve.

The rest of the experiments are conducted in a similar manner. A sample program is annotated with the directives, evolved, and then compared with the best hand-coded result. The rest of this section discusses methods of improving the convergence time and tests on more complex environments and source code.

4.3 Convergence Properties

A principle part of this research is to determine a set of optimal parameter settings for the GA that will quickly and consistently evolve nearly optimal programs. However, there is no single best set of parameter settings; the best parameter settings are dependent on the the specific problem. This section examines the relationships between the different parameter settings and finding an optimal result.

4.3.1 Non-optimal Convergence Goal

First, it is important to more closely examine the goals of convergence for the genetic algorithm. The evolutionary system is effective because it is probabilistic, yet the randomness of the system can also serve to undermine it. It is possible to encourage so much randomness in the system, that the evolved results are random as well, with no guarantee that such a result is the best possible. In fact, any randomness in the system disallows any guarantee of finding an optimal result, yet it would be comforting to know that the system will find a nearly optimal result with very high probability. The convergence goal is to choose a set of evolutionary parameters that with high probability will evolve an optimal result.

This goal can only be accomplished by properly balancing the forces of convergence and randomness in the evolutionary system. Optimizing this balance is not within the scope of this research. As a researcher, tuning the genetic algorithms is a seductive pursuit. It is tempting to spend weeks changing parameters to get closer and closer to achieving an optimal result on every test. This research is not specifically on genetic

algorithms, so a less ambitious goal was used. The genetic algorithm should with very high probability (99%) exceed the performance of traditional techniques. This goal is short of the goal of achieving the best possible performance, but since the best possible performance is unknown, it is a more measurable goal. However, it is hoped that in a run of the successive identical trials, most (90%) will have best evolved solutions within 10% of the best solution found over all the trials.

Given the set of parameters used in the initial trials, this goal is not difficult to attain for easy problems. Figure 4-2 shows a graph of population statistics per generation for ten evolutionary run trials. In these trials, a population size of 60 was used for 80 generations with a replacement size of 30. The three data series are the worst individuals in the population, the average of all individuals in the population, and the best individual in the population. Each of these quantities was found for each trial, and then averaged for the ten trials graph.

The GA rapidly finds the optimal solution. In most cases, by the 30th generation the GA has found the optimal result. In ten generations, the average fitness of the population improves by a factor of two, and by 50 generations nearly all the individuals are within 10% of the best individual.

In addition to the rapid convergence, all of the trials tend to result in similar best individual scores, as shown in the graph in figure 4-3. This chart compares the best performing individual in the population for each trial. Over all the trials, the best of that set, the worst, and the average value are graphed. At the end of the run, neither varies by more than 10% from the average. The chart in figure 4-4 compares the difference between the best time and the worst time in the 10 trials, over each generation. The difference is much larger in early generations, since the initial populations are randomly generated. Eventually the convergent properties of the GA drive the difference to a much lower level.

For an easy distribution, the evolutionary system performs consistently within accordance of the convergence goal. More difficult tests are detailed below, but in all cases if the GA were able to find a solution better than the traditional techniques, then multiple runs of that same trial always result in a best evolved result better than

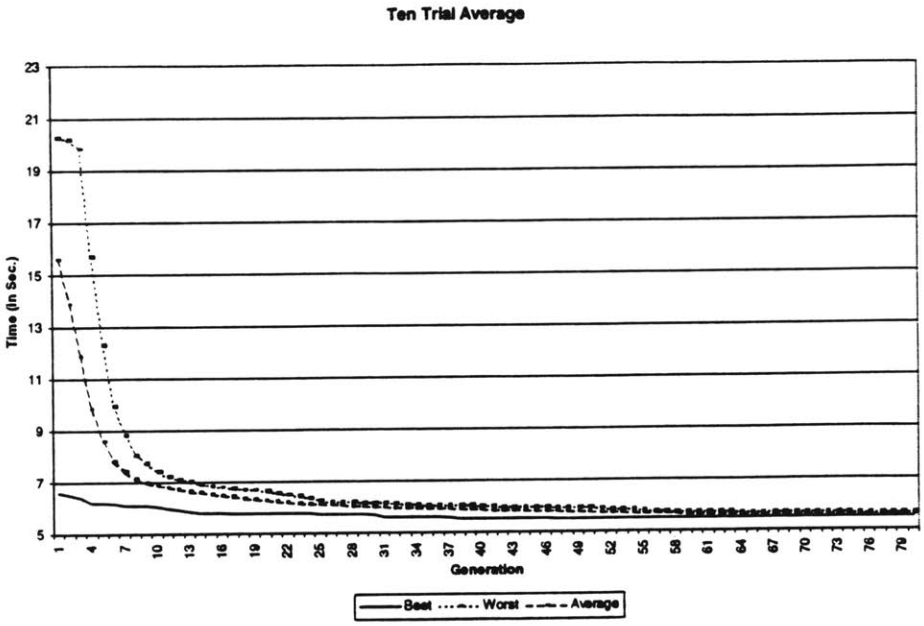

Figure 4-2: Ten Exponential Distribution Trials, Entire Populations Graphed

traditional techniques. Further more, no more than a 10% variability in the fitness of the best evolved solution was ever found in the tests. The parameter settings are detailed in the next sections.

4.3.2 Mutation Rate Has No Effect

The mutation rate controls the probability of mutation during the creation of a new individual. After the initial populations are generated, mutation serves as the primary source of randomness in the population. As stated before, mutation is rarely used in genetic programming, since alleles rarely become extinct.

The mutation rate had little effect on the efficiency of the run. The graph in figure 4-5 shows data points for a 50 trial test. Each trial was identical, except for the mutation rate. The graph compares the total number of mutations that occurred during the trial with the score of the best individual in that trial. Regression analysis of this data can find neither a line nor a polynomial curve that fits the data and has

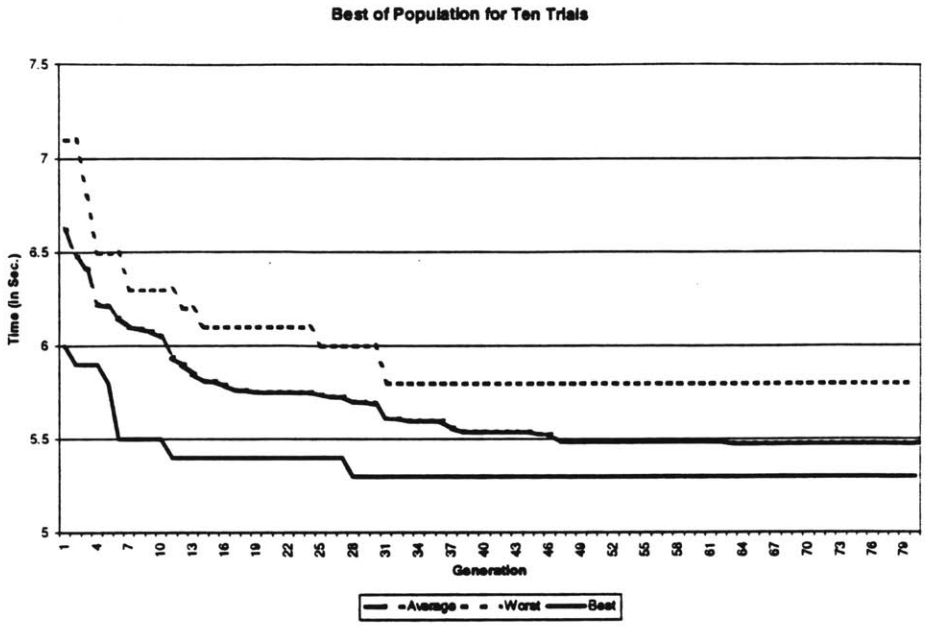

Figure 4-3: Ten Exponential Distribution Trials, Best Individuals Only

a statistically significant t-stat value. [8] The mutation rate and the score of the best individual in the population are unrelated.

4.3.3 Large Populations for Many Generations Produce Good Results

The population size and number of generations tests demonstrated a clear result: larger populations and evolutions that last for more generations produce better results. To test the effects of these variables on the performance of the GA, 48 separate evolution runs were executed, each with a different population size and number of generations. All tried to optimize the same program, the computation bound program discussed in section 4.4.3. This program is difficult to optimize, so the inadequacies of the smaller test are more pronounced. An easy distribution would have simply shown all the results having nearly the same performance.

The results of this experiment are shown in the chart in figure 4-6. The X axis is

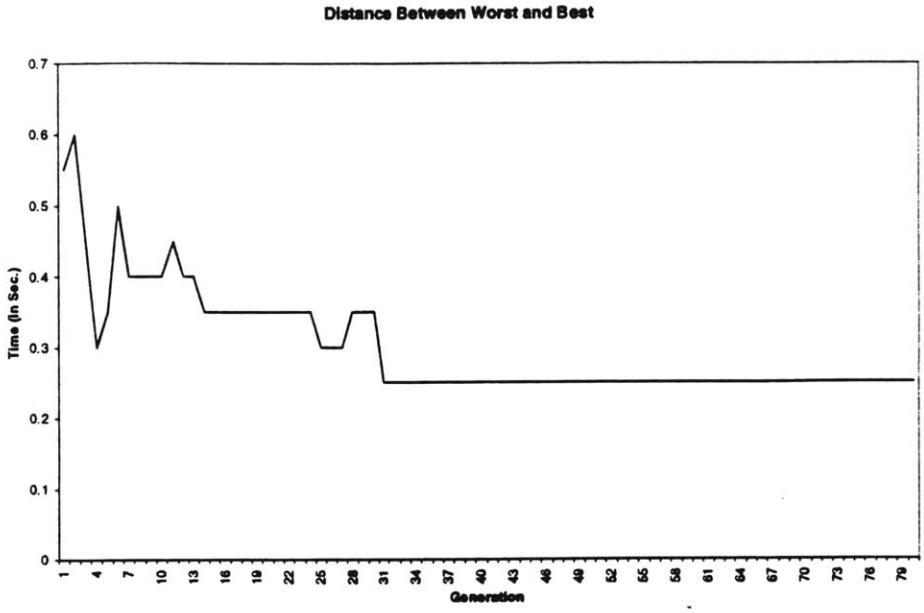

Figure 4-4: Difference in Best and Worst Individuals

the number of generations and the Y axis is the size of the population. The Z axis is the time of the best result, so a lower value reflects better performance. Increasing both the population size and the generation number improves the performance. Increasing just one variable is not sufficient. Large population sizes for a short number of generations and small populations for many generations do not always produce good results.

4.3.4 Replacing Half the Population is Ideal

The goal of the replacement number experiment is to determine the relationship between the number of replaced individuals per generation and the optimal score generated. For a population of 40, the number of individuals replaced was varied from 5 to 37. Each test executed for 60 generations, and the test program was a simple exponential distribution. The score of the best individual in the final population was recorded for each test.

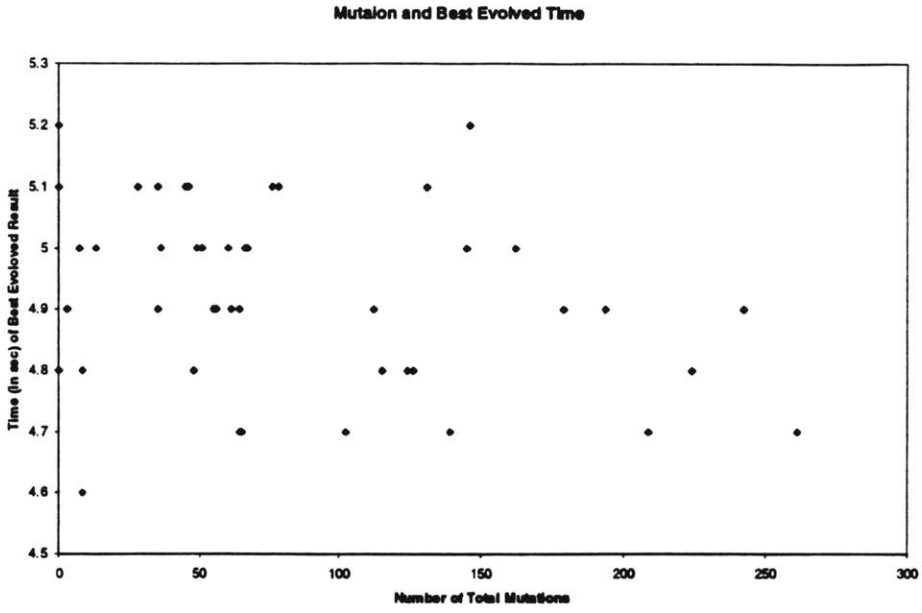

Figure 4-5: Mutations and Best Evolved Result

The chart in figure 4-7 shows that more replacements yield better individuals in the final population, but the correlation is small. Further statistical analysis shows that with both few replacements and a large number of replacements, the variance from the mean over all replacement tests is much higher. The probability of having a very bad result is higher for extreme replacement numbers, although this is somewhat masked by better average results for higher replacements. For consistent performance, a number of replacements nearly equal to half the population size is ideal.

4.4 Other Test Environments Results

This section describes tests using real application code and more complicated test environments. Each test description includes a table detailing the results. These tables are self-explanatory, except there are some issues with the timing measurements. The times reported are not the total running time for the program, instead the times are the sum of all the execution times of all the loop entities in the code. This cap-

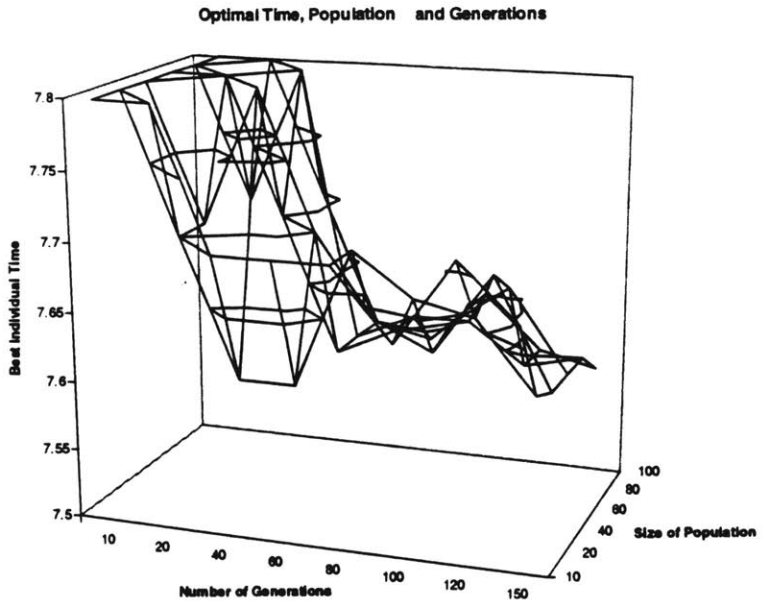

Figure 4-6: Population Size, Generation Number, and Optimal Score

tures all of the parallel regions and any additional time required by the parallelism, but does not include serial code. This measurement directly compares the scaling of the evolved code and the traditional approaches without the distortion of serial code that does not scale. The serial time is measured by compiling the code with no parallel constructs and then executing it on a single processor. The evolved code and traditional scheduled code are timed in the same manner. They are executed at least ten times (some tests have more variability and are executed more), and the averages are reported in the table. Both the evolved code and the traditional scheduling techniques are compared, with the best traditional scheduling score serving as a control for the experiment.

Throughout this discussion of other test environments, the words evenly and equally are used to describe two very different types of distributions.

Evenly is used when every thread receives the same number of iterations.

Equally means that every thread has the same amount of work.

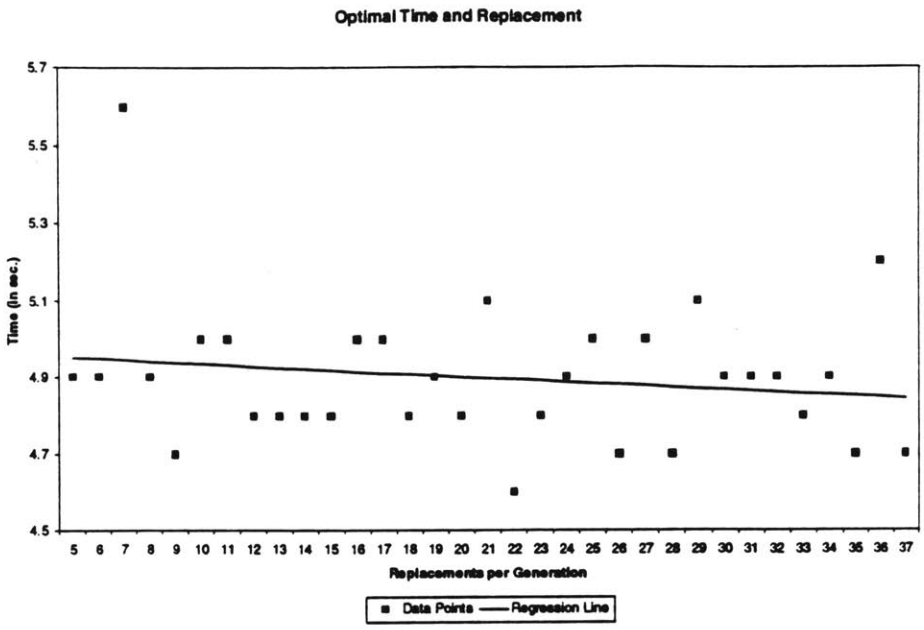

Figure 4-7: Replacement Number and Optimal Score

In the case of a uniform work distribution, evenly and equally are equivalent; however, in any distribution where each iteration has a different amount of work, these two have very different performance results. The goal of this work is to evolve distributions that are equal across the system, not even.

4.4.1 Shared Memory Applications Have Varied Results

While the initial tests demonstrated that the genetic algorithm evolves optimal solutions for a variety of test distributions, it was not clear that it would perform so well for real application code. While the test code had only a few variables that had no data dependence, real application code is rarely so benign. Three programs were chosen from the body of scientific numerical applications to serve as test codes. Programs important to the scientific community were chosen because they are one of the primary groups who could benefit from this work. The target architecture in these tests is the same as in the initial trials, a 4 processor symmetric multiprocessor

machine. All processors are equally fast and have equally fast access to the memory.

The three selected programs are a Monte Carlo simulator, a N-body simulator, and a Fast Fourier Transform (FFT). The Monte Carlo simulator is "embarrassingly parallel," [30] the individual iterations have no data dependencies. The single parallel loop generates a random number and places the results in an appropriate data location. The variables have no data dependence within the loop or across loop iterations. Also, each individual iteration does little work, so fine-grain parallelism is possible. The N-body simulator is more complex; it has 4 parallelizable loops and some with data dependencies requiring locks and critical sections. The FFT is the most complex of the three, with nested loops, data dependencies, and non-obvious parallelism.

Each of these codes were in their original form serial programs. The KAP^{TM} [21] parallelizing pre-processor was used to parallelize them. KAP annotates the serial code with parallelizing directives and additional code to handle shared and private variables. KAP first identifies parallelizable loops by analyzing the data dependencies and marking the variables that must be shared and those that must be private. With this knowledge, KAP inserts the appropriate parallel directives. These directives were then changed to APR directives, and then the programs were sent to the evolutionary system. These steps were rarely free from errors— often KAP would fail to recognize potential parallelism due to complicated data dependencies. These conflicts were solved by hand modifying the parallel sections. Constructing the hand-coded control programs took an average of 8 hours, which is roughly equal to the amount of time the GA required to find its solutions for these tests.

Each of these three tests is discussed in more detail in the following sections.

Monte Carlo

A Monte Carlo simulation is an estimation technique that has been applied to many different problems in a variety of fields. The simulation consists of many probabilistic trials, the results of which are tallied and used to determine some unmeasurable but testable quantity. [40] A simple example would be to estimate π by throwing darts at a circle of unit diameter enclosed in a unit square. The area of the square is

known, so by tallying the proportion of darts which land in the circle compared to the square, the area of the circle can be estimated, and from that π can be determined. The probabilistic law of large numbers guarantees that as more trials are used, the estimate is more likely to be closer to the actual value. [8]

The Monte Carlo simulation generates millions of independent trials, each of them completely parallelizable. The result is an even distribution that perfectly scales when using traditional techniques. Figure 4-8 shows the results of an evolved test.

Test Name	Monte Carlo
Target Architecture	4 Processor SMM
Population Size	60
Generations	80
Running Time (hours:minutes)	
Serial Time (sec)	31.5
Best Evolved (sec)	7.8
Dynamic Time (sec)	7.8
Static Time (sec)	8.1
Guided Time (sec)	8.1
Best Evolved Scaling	4.0
Best Control Scaling	4.0

Figure 4-8: Monte Carlo Test

The best distribution algorithm evolved is of the form

$$k_1 x + k_2 + sin(x + k_3)e^{k_4}$$

where x is the current iteration and k_{1-4} are constants. The distribution is fairly simple, owing to the nature of the regular scheduling required by the problem. The work is evenly distributed with an offset of k_2, and then a small error term changes some of the balanced distribution, possibly with little effect or potentially to take advantage of some poorly understood system performance situations.

This test essentially shows that in the case of a simple work distribution, the evolved answer does not introduce enough overhead to prevent it from scaling just as well as the traditional techniques. Also, the GA was able to find a solution quickly. The first generation found a 7.9 second algorithm, a 7.8 second result was found

in generation 5, and the final best result was found in generation 23. For simple distributions, the GA quickly achieves the same level of performance as the best traditional techniques.

N-body Simulation

An n-body simulation projects the movement in 3-space of an arbitrary number of particles interacting in various gravitational, electrostatic, and electromagnetic ways. [14] The data used for these runs included 10,000 separate particles.

The n-body simulation code is not a good application of the evolutionary system. The program consists of four parallelizable loops, three of which do almost no work, and one nested loop where almost all of the computation takes place. An initial trial attempted to schedule all four of the loops. The three low-computation loops had running times that were less than the reliable granularity of the timing mechanism. Without any selective pressure, the algorithms grew to be extremely large expressions. These large expressions would take marginally more time to execute, but this level could not be detected, so the expressions continued to grow larger. Eventually, the expressions got so large that they could no longer be optimized by the compiler, so the running times went from less than a hundredth of a second to hundreds of seconds. The population had become filled with these bloated expressions, such that almost any combination would yield an expression that exceeded the optimizing compiler's capabilities. The run stalled after only a few generations. Eventually the large expressions would be filtered out of the population, but not until after much wasted computation.

The second attempt tried only to optimize the single loop which accounts for almost all of the running time of the simulation. This test also proved to be unsuccessful. The optimized loop is within an outer loop. The outer loop can not be similarly parallelized because it contains inherently serial code needed to parallelize the inner loop. The inner loop is executed thousands of times, and does a little computation and many memory loads and saves. The results from parallelizing this loop are in figure 4-9

Test Name	N-body Simulation
Target Architecture	4 Processor SMM
Population Size	60
Generations	80
Running Time (hours:minutes)	12:14
Serial Time (sec)	7.5
Best Evolved (sec)	12.6
Dynamic Time (sec)	13.2
Static Time (sec)	2.2
Guided Time (sec)	7.2
Best Evolved Scaling	0.6
Best Control Scaling	3.4

Figure 4-9: N-body Simulation Test

The best results for both the evolution system and the dynamic scheduler were significantly worse than the serial execution time. Both of these results were with only executing on one processor; multiple thread execution caused times as high as 44 seconds in the evolutionary system and 55 seconds in the dynamic approach. While these two techniques were not effective, the static scheduler performed well. The static scheduler may provide a non-optimal distribution, but it requires almost no overhead. In the case of the dynamic and evolved solutions, the overhead of adding additional function calls caused a significant performance degradation.

Fast Fourier Transform

The Fast Fourier Transform is an algorithmic technique for performing the Fourier Transform in $\log(n)$ independent computations. [30] Like the other applications, this program was originally serial code, so it was first converted to a parallel version with the KAP preprocessor. The code that KAP generated was then altered to improve parallel performance. The KAP directives were then replaced by APR directives for the evolutionary system.

Much like the N-body simulation, this application was not well suited to the genetic algorithm. The code required many serial memory accesses, and has loop iterations that have little computational work. Unlike the N-body simulation, this test performed reasonably well. Figure 4-10 shows the results. Although the overall

Test Name	FFT Application
Target Architecture	4 Processor SMM
Population Size	60
Generations	80
Running Time (hours:minutes)	10:32
Serial Time (sec)	9.8
Best Evolved (sec)	7.1
Dynamic Time (sec)	14.6
Static Time (sec)	7.3
Guided Time (sec)	8.5
Best Evolved Scaling	1.3
Best Control Scaling	1.4

Figure 4-10: FFT Application Test

scaling was not good, it was able to extract the same level of parallelism as the control methods. In both the N-body simulation test and in this test the dynamic time was worse than the serial time because the loop iterations do not have enough work to hide the overhead of scheduling them. While in the N-body test the evolved solution also had this property, in this test it was able to vary the granularity, and move the parallelism to an outside loop. The thread teams allowed the system to serialize the inner loops, and distribute the more work intensive outer loops. Combined with a more efficient scheduling algorithm, the evolved solution was able to perform better than the best static time.

4.4.2 The GA Optimizes Mixed-mode Parallelism

Mixed-mode parallelism refers to using both MPI and OpenMP protocols to match the granularity of the parallelism to the communication protocol that has the appropriate amount of latency and bandwidth. Mixed-mode parallelism is used in cluster configurations, where each processor does not have access to all the memory of the system, and sharing data between nodes in the cluster entails a definite performance cost. By varying the work distribution and distribution algorithms, the GA should be able to dramatically improve the performance over traditional techniques, since the cluster configuration is more complex than a simple shared memory machine. The

cluster in this case consists of two 4 processor shared memory machines from the earlier tests, connected with a low, but variable latency memory channel interconnect. Accessing the memory channel is approximately 15 times slower than the main memory, and approximately 150 times slower than the cache. By distributing a large problem into smaller pieces that fit easily into cache, the GA may be able to achieve superscaling speed-ups by using the memory hierarchy to its advantage.

To test this hypothesis, a test program was developed. Nearly all the work in the program is computation over a 20x100 array. As the program moves through the array, each element of the array requires exponentially more work, according to the FORTRAN code snippet:

```
do i = 1,20
  do j = 1,100
    b(i,j) = kidint(exp(real(i)*real(j)/100)) * 3
  end do
end do
```

The variable latency of the communication network in the cluster adds an additional complication. The timing of a piece of code is highly variable. On the first attempt to evolve an optimal solution, the GA appeared to have found a good solution. When the best algorithm was actually tested, it was discovered to perform rather poorly on average and only occasionally perform well. Since the GA executes so many tests, it is possible for such an algorithm to perform well once, secure a spot in the population, and then pollute the gene pool with its actually inferior genes. One attempt to remedy this problem was to replace the entire population in each generation, so bad single test results would be multiply tested and removed. This approach was only somewhat successful, since replacing the population further slowed convergence. Another approach was to multiply execute the test program and use the average as the fitness. This approach was more successful. Though it is possible for a program to be incorrectly timed and have inaccurate fitness in all trials, it is much less likely. Each program was executed three times, which seemed to adequately

balance the benefit to fitness accuracy and the increase in computation time.

Test Name	Mixed-mode
Target Architecture	2 node, 4 Processor SMM cluster
Population Size	80
Generations	60
Running Time (hours:minutes)	47:16
Serial Time (sec)	54.5
Best Evolved Mean (sec)	6.7
Dynamic Time (sec)	15.5
Static Time (sec)	40.4
Guided Time (sec)	13.3
Best Evolved Scaling	8.1
Best Control Scaling	4.1

Figure 4-11: Mixed-mode Parallelism Test

The table in Figure 4-11 estimates the evolved solution running time at 6.7, however some tests executed in 6.2 seconds. Others took 7.5 seconds. Over 30 executions, the mean was 6.7 and the standard deviation 0.32 seconds. The scores for the control runs are also averages, but they tended not to vary as much as the evolved solution.

In this test, the GA achieved a running time 1.9 times as fast as the traditional techniques and achieved super scaling. The algorithm converged on a thread team that distributed the outer loop between the two nodes, and the inner loop within each node. This division led to greater locality of data, leading to better cache performance. The algorithms themselves are difficult to analyze. Figure 4-12 contains the distribution algorithm from the outer loop algorithm. The inner loop algorithm

```
((((((zSD((7 + 0), 0) - coefEx(CR, trigSin(7, 17))) * zSD((1 + CR),
7)) - (trigSin(CR, (1 - MX)) * (coefEx(zSD((coefEx((coefEx(coefEx(((5
+ trigSin(zSD(trigSin(CR, (1 - MX)), CR), 2)) * 17), 0), (5
+ coefEx(CR, trigSin(7, 17)))) * zSD(trigSin(2, zSD(PR, 7)),
coefEx(zSD(13, 7), 17))), zSD(MX, 1)) + 13), (zSD((2 - PR), 17) +
zSD(MX, PR))), 11) + coefEx(coefEx(1, (5 + 5)), 0)))) - (1 + 7)) *
(1 + CR)) - (trigSin(CR, (1 - MX)) * (coefEx(zSD((1 + CR), 1), 11) +
coefEx(coefEx(1, (5 + 5)), 0))))
```

Figure 4-12: Optimal Distribution Algorithm for Mixed-mode Test

is even more complicated. Even further complicating the situation is the state of

the thread team. The outer loop is actually distributed among 4 processors, and the inner loop is distributed among blocks of two processors within the machines. Both algorithms effect the distribution to the individual processors, so the algorithm makes no clean division between nodes. The result is that each algorithm appears to have a distribution that would not be nearly optimal, but when all the pieces are assembled in concert, the system achieves super scaling. This result is a demonstration of the capacity of the symbiotic genetic algorithm. By capturing the interaction between the loops, the GA was able to far surpass traditional techniques.

4.4.3 Large Cluster Execution

A further test of the evolutionary system is to distribute a complex workload across a large parallel machine. Two tests, one memory bound and one computation bound were designed for an eight node, four processor per node, low latency crossbar interconnect system. This machine differs from the previously described cluster in several ways. First, the individual nodes are equipped with faster processors, larger caches, and more memory bandwidth. Second, the inter-node communication is handled by a low latency, high bandwidth crossbar. These features make scaling on a considerably larger system feasible. Also, since the interconnect latency is nearly constant, it was not necessary to run the tests multiple times. Each of the two types of tests are discussed in the subsequent sections.

Memory Bound

None of the previously discussed tests were memory bound; they all simply required a distribution of the computation. Intuitively, it is not clear how the GA could also improve the performance of memory bound computation. The individual processors share the same memory bandwidth, so moving computation to a different processor would not reduce the amount of memory resources used. However, the genetic algorithm can modify the way the entire program runs across the cluster. By distributing work that is memory intensive across many machines, and improving data locality

and cache performance, the GA can use the memory hierarchy to achieve superior performance enhancements over traditional techniques.

The memory bound test is a matrix multiply of two large matrices. The algorithm consists of two parallelizable outer loops that iterate over the columns and rows, and an inner loop which performs the actual multiplication of the selected rows and columns. When framed in this manner, the algorithm does not have any data dependencies, but it does severely tax the memory system. [13] For every multiplication, two memory accesses are required. Also, the matrices are far too large to fit in cache, and since the specific architecture of this experiment has two-way, set-associative cache, the cache is completely replaced with each multiply for matrices of this size.

Two different matrix sizes were tried for this test: 128x20000 and 1010x1010. The two different sizes had dramatically different results, but also shared some similarities. Neither test was able to scale perfectly with the number of processors. Individual node memory bandwidth was likely a large part of the problem. The test would perfectly scale up to eight processors, if each processor was on a different node. Running multiple threads on a single node did not only fail to perfectly scale, but at 32 processors, the test performed worse than on just eight processors. Executing multiple threads on a single node caused bus saturation and excessive dirtying of the write-through cache as multiple threads worked on the same cache lines. Also, both tests had difficulty finding distributions with better than 8 times speed-up. Even near the end of the generations, only a small fraction of the final population had any members that were better, so the population never converged. It is likely that finding such distributions is difficult, and requires exploitation of very sensitive implementation details of the hardware and optimized source code.

The 128x20000 matrix test attained improved performance in an unintuitive way. The thread team distributed the outer loop to the eight nodes, and the inner loop was distributed within each node. The outer loop evenly distributed the work in an orderly fashion, but the inner loop distributed the work serially. Only one thread computed the inner loop iterations. The GA recognized the saturation properties,

Test Name	Matrix Multiply, 128x20000
Target Architecture	8 node, 4 Processors per SMM Node
Population Size	80
Generations	50
Running Time (hours:minutes)	8:03
Serial Time (sec)	33.4
Best Evolved Mean (sec)	2.1
Dynamic Time (sec)	2.8
Best Evolved Scaling	15.9
Best Control Scaling	11.9

Figure 4-13: Matrix Multiply 128x20000 Large Cluster Test

and discovered that the serialization worked better than parallelization for the inner loop. The test still achieved nearly 16 times speed-up, most likely due to cache effects arising from the outer loop distribution. Curiously enough, changing the serial inner loop distribution to distribute to multiple processors slowed the test, as more threads competed for scarce memory resources.

Test Name	Matrix Multiply 1010x1010
Target Architecture	8 node, 4 Processors per SMM Node
Population Size	150
Generations	80
Running Time (hours:minutes)	54:21
Serial Time (sec)	95.3
Best Evolved Mean (sec)	3.6
Dynamic Time (sec)	8.0
Best Evolved Scaling	26.5
Best Control Scaling	11.9

Figure 4-14: Matrix Multiply 1010x1010 Large Cluster Test

The 1010X1010 test performed differently than the other matrix size, but the difference in result may have little to do with size of the arrays and more to do with the larger population. Here the GA attained a 26 times speed-up. The optimal thread team was again the same as before, with the outer loop distributed to the eight nodes, and the inner loop distributed within the node. The distribution algorithms were different. The outer loop distributed evenly across all nodes, but the inner distribution algorithm distributed to all four processors. The outer distribution can

be reduced to the simple expression

$$30 - CR/34$$

which implies the iterations are distributed in large blocks. Also, the inner distribution can be expressed as

$$126 + PR/8$$

Here PR is the additional loop parameter, which is passed into the algorithm for the inner loop as the iteration of the outer loop. The difference is that each inner loop is not distributed. All of the iterations of the inner loop go to the same thread, but when a different outer loop iteration is sent to that node, it is computed by by a different thread. The outer loop distributes to each node a block of iterations. Each iteration is computed by a different thread, so the threads are not working on the same cache lines and causing coherency problems. This interdependence is another example of the advantages of the symbiotic genetic algorithm. The two distribution algorithms and the thread team are able to find a previously unknown and efficient task scheduling. By constructing a test with a larger population and more generations, the genetic algorithm found a result that is better than traditional techniques.

Computation Bound

The computation bound experiment is similar to the experiment on the two node cluster, except a 200x100 array was used rather than a 20x100. The same exponential distribution was also used, except some constants were changed to provide a running time that was manageable for the test. Choosing these constants initially proved to be challenging. If the array was too small, then the GA would quickly decompose the problem so it ran entirely in cache and achieved a 100 times speed-up. Such a test was not desirable, since the timing mechanism does not have good enough granularity to measure such fast programs. Eventually, code was inserted into the loop which invalidated cache lines and made such a result impossible. This addition caused only a minimal amount of memory access, so the problem remained primarily

computationally bound.

Test Name	Cluster Computation Bound Test 1010x1010
Target Architecture	8 node, 4 Processors per SMM Node
Population Size	150
Generations	80
Running Time (hours:minutes)	47:39
Serial Time (sec)	64.0
Best Evolved Mean (sec)	3.7
Dynamic Time (sec)	6.1
Static Time (sec)	12.0
Guided Time (sec)	6.0
Best Evolved Scaling	17.3
Best Control Scaling	10.1

Figure 4-15: Large Cluster Computation Bound Test

The GA was not able to find and optimal distribution that perfectly scaled, but judging by the performance of the best traditional technique, the problem is not easily distributed. The nature of the problem may not allow any better scaling. The evolved distribution algorithms are difficult to analyze; the outer loop distribution algorithm has 455 nodes. Since the distribution is suitably complex, it is appropriate to have a complex distribution algorithm.

4.4.4 The Evolutionary System is Not Data Dependant

One of the primary criticisms of this type of work is that the optimal work distribution evolved by the genetic algorithm is data dependent. [27] Some would claim that the gains found over traditional techniques are only due to exploiting irregularities in the test set, and if the application was tested with the same algorithm and a different data set, the results would be significantly worse. This claim seems intuitive, but is unfounded, so it merits further investigation.

If the genetic algorithm simply optimizes for the test data, then adding randomness to the data set would leave the GA incapable of doing any better than traditional techniques. To test this assumption, tests were created that added randomized data in three ways, added noise to a well known distribution, randomly changing the ori-

entation of the data, and changing the size of the data. If the changes to the data fundamentally change all the performance characteristics of the original program, then a genetically evolved solution will not be helpful. Most programs have similar performance characteristics over a wide range of data, though the pattern is often difficult to discern. The purpose of these tests is to simulate that model of program use, and determine if the evolutionary system can find and exploit such patterns. The results of these tests are detailed in the following sections.

Noisy Exponential Distribution

This test consists of adding noise to the exponential work distribution used in the first initial trials. Half of the total work was in the exponential distribution, the other half was randomly distributed from a uniform probability function. The resulting distribution resembles an exponential distribution, but has a random amount of distortion in each run.

The population size and number of generations are larger in this test compared to the initial runs. This problem is more challenging compared to the noiseless test, so the GA should have more computational capability. Also, each of the programs were executed three times and the resulting score averaged to measure the fitness. This redundancy should avoid the problem of evolving a single program that performs well for a certain set of added noise, but poorly in other cases.

Test Name	Random Noise
Target Architecture	4 Processor SMM
Population Size	80
Generations	80
Running Time (hours:minutes)	
Serial Time (sec)	18.4
Best Evolved (sec)	4.5
Dynamic Time (sec)	5.0
Static Time (sec)	7.6
Guided Time (sec)	4.8
Best Evolved Scaling	4.0
Best Control Scaling	3.8

Figure 4-16: Added Random Noise Test

The genetic algorithm was able to develop an distribution algorithm that filtered through the noise and properly distributed for an exponential algorithm in the general case. The optimal algorithm found in this test is

$$k_1 - k_2 \sin(k_3 e^x) - x$$

For the smaller iterations, the $k_3 e^x$ term dominates the distribution, but for large values of x, the maximum value of the e^x term is capped to prevent overflow. This term becomes constant, and the distribution is dominated by the $-x$ term. The result is the larger iterations are evenly distributed, but since the larger iterations are exponentially increasing, the distribution is ragged. The smaller iterations are not evenly distributed, but instead the $\sin(k_3 e^x)$ term distributes the iterations in a way to equalize the total work.

The distribution found with added noise is similar to the distributions in the initial run, except the function governing the distribution of the smaller iterations is different. Despite the addition of noise, this test was able to achieve perfect scaling, and surpass the best traditional approach by 20%, nearly identical to the findings in the noise-free tests. The addition of the random noise has little effect on the performance of the GA, despite noise accounting for half of the total work distribution.

Random Orientation

The random orientation test uses an exponential distribution, but with probability 0.5 the exponential distribution will be decaying instead of growing. Both possibilities have the same amount of work, but the decaying exponential is just the mirror image of the growing exponential used in other tests.

As in the previous test, a larger population and generation number were used in this test. Also, all programs were executed three times and the scores averaged to determine the fitness.

The generated distribution algorithm did not develop a specialized mechanism for dealing with the two different exponential cases. Instead, much like in nature, the GA

Test Name	Random Orientation
Target Architecture	4 Processor SMM
Population Size	80
Generations	80
Running Time (hours:minutes)	
Serial Time (sec)	18.5
Best Evolved (sec)	4.6
Dynamic Time (sec)	4.8
Static Time (sec)	6.7
Guided Time (sec)	4.8
Best Evolved Scaling	4.0
Best Control Scaling	3.9

Figure 4-17: Random Orientation Test

found a simpler way. The algorithm is not effected by the orientation, even though a boolean that identifies decaying distributions was provided to the algorithm as a parameter. Instead it distributes the iterations in an even manner. However, it is rare for all of the growing exponentials do be together in the distribution; the two types are interspersed. At the end of a growing exponential, some threads are still completing the previous work, while others are idle. These idle threads continue to the next iteration. If the next iteration is the opposite of the previous one, then the threads are given a large amount of work to do. Although the amount of work in a single exponential distribution is not equally distributed, because the two exponential types are symmetric, the total work is equally distributed. If the threads have to stall after completing each exponential and wait for all the loops to finish, the performance is much worse. By generating a distribution that was not optimal in either specific case, the GA was able to evolve a distribution that worked well in the overall system.

The evolutionary system was able to find a result of perfect scaling, despite not optimizing for the individual cases. Instead the GA found a surprisingly simpler way to achieve the optimal performance.

Random Loop Size

The random loop size experiment tests the evolutionary system's ability to find optimal solutions for a common programming environment—a nested loop that executes

for a different number of loop iterations with each iteration of the outer loop. In the test program, the outer loop randomly selects a number of iterations for the inner loop. The selected number comes from a previously generated set, so the actual total number of loop iterations does not change, just the order in which they are presented to the inner loop. Each program must have the same amount of total work in order to compare program fitness values. The inner loop is not just a uniform distribution. To make the problem difficult, the inner loop has an exponential distribution. In order to adjust to the different loop sizes, the scheduling algorithm is provided with the maximum number of iterations for the loop as a potential parameter for the evolved distribution.

Like the previous random data tests, more generations and a larger population was used. Also, each program was executed three times in order to prevent premature convergence on an inefficient solution. The threads are not required to wait for all the iterations of a loop to complete before continuing to execute code.

Test Name	Random Loop Size
Target Architecture	4 Processor SMM
Population Size	150
Generations	80
Running Time (hours:minutes)	
Serial Time (sec)	17.7
Best Evolved (sec)	4.5
Dynamic Time (sec)	4.7
Static Time (sec)	10.6
Guided Time (sec)	4.8
Best Evolved Scaling	3.9
Best Control Scaling	3.8

Figure 4-18: Random Loop Size Test

The results for this experiment were also surprising. At first glance, the evolved distribution appears inordinately complex, and graphing the results reveal that the iterations are distributed evenly. A closer look reveals the following equation:

$$k_1 + \mathcal{F}(MX) + x$$

where $\mathcal{F}(MX)$ is a complex function of the maximum iteration for the loop. This function determines an offset for the even distribution. After threads finish all the iterations assigned to them in one loop, they continue to the next sized loop, with an offset. This prevents the same threads from doing all of the work in all of the different sized loops. The function $\mathcal{F}$ is suitably random such that over many differently sized loops the work is equally distributed.

In all three cases of randomness in the performance characteristic of a program, the evolutionary system was able to identify a pattern and develop an optimal algorithm for that situation. In some cases, such as the noisy exponential, the GA evolved into the same distribution as the non-random case. In the other situations, the GA did not evolve individually optimal algorithms, but instead it found distributions that were optimal over the entire execution of the program. In all cases, the GA did not prematurely converge on data dependent solutions, and instead found widely applicable algorithms.

4.5 Target Architectures

This section is dedicated to discussing the power of the evolutionary system to adapt solutions to a wide variety of different target architectures.

4.5.1 Altering the Target System Causes Adaptive Behavior

The evolutionary system can converge to a nearly optimal result, and it can continually alter the population to adapt to changes in the target system. In this test, the target architecture is a four processor shared memory machine. After 20 generations, a high priority, computationally intensive process was launched on the machine. This program constantly used one of the processors, and rarely yielded any time. At 20 generation 40, the added program was killed.

Figure 4-19 shows the population scores for this test. During the first 20 generations, the evolutionary system rapidly converged toward a four processor solution. When the one processor was removed, the entire population suffered a performance

loss. In subsequent generations, it began evolving toward a three processor solution, and the best individuals avoided the busy processor. Much of the performance loss was recaptured, and the best individual achieved perfect scaling for three processors. After the next 20 generations passed, the fourth processor became free, and some of the other individuals began to perform better than the three processor optimized individuals. Soon, the best individuals in the population returned to perfect scaling for four processors. Not only is the evolutionary system robust enough to optimize for different target architectures, but also it can also optimize over a changing system.

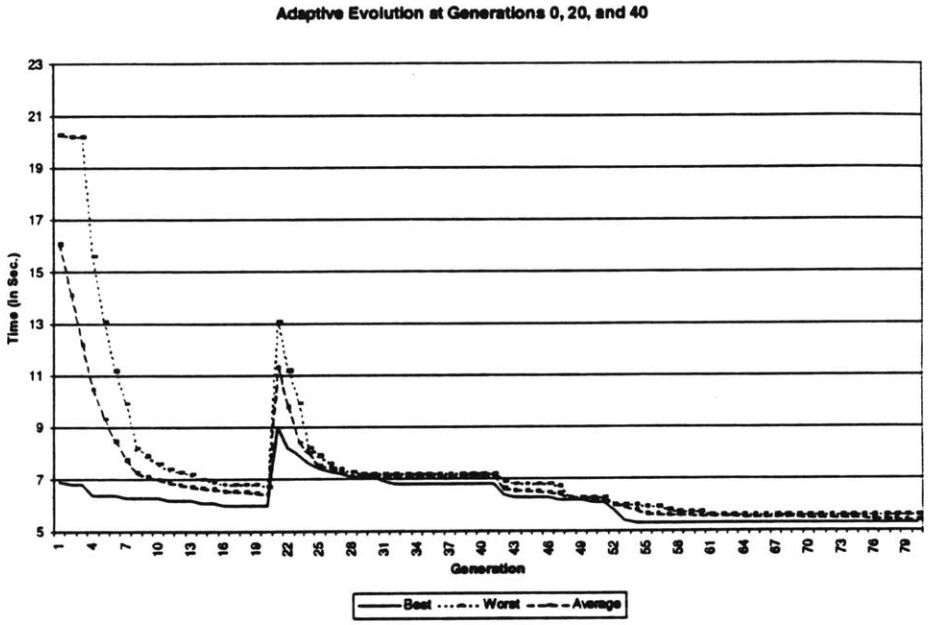

Figure 4-19: Population Scores with Changing Architecture

4.5.2 GA Adapts to Heterogeneous System

All of the previously described tests have completely symmetric processors. While the communication latencies and bandwidth from each processor varied if they were on different nodes, all processors were computationally identical. In this test, two of the nodes in the cluster have slower processors with less memory bandwidth, and the

other 2 nodes have processors that perform twice as fast.

This test used the same exponentially increasing work distribution used in the computation bound test in section 4.4.3. The results are described in figure 4-20. In the earlier computation bound test, the best result achieved was scaling of 17.9 on 32 processors. The communication systems on the earlier test were much better, yet this heterogeneous test still achieved scaling of 8.8 on 16 processors, which is slightly worse yet very similar performance. The distribution algorithms distribute most of the iterations evenly, yet the smaller iterations are distributed to the faster nodes, equalizing the total work. Without knowledge of the difference in the nodes, the evolutionary system adapted in a away that took advantage of this performance characteristic.

Test Name	Heterogeneous
Target Architecture	4 node, 4 Processor SMM cluster
Population Size	150
Generations	80
Running Time (hours:minutes)	
Serial Time (sec)	66.5
Best Evolved Mean (sec)	7.6
Dynamic Time (sec)	10.8
Static Time (sec)	13.7
Guided Time (sec)	9.9
Best Evolved Scaling	8.8
Best Control Scaling	6.7

Figure 4-20: Heterogeneous Test Environment

Chapter 5

Conclusions

5.1 Overview

The primary focus of this research is to develop an evolutionary system capable of producing a nearly optimal work distribution for a specific target parallel architecture. For many types of parallel programs, the evolutionary system was largely successful. In most of the test programs, the evolved solutions consistently outperformed traditional techniques on a variety of hardware platforms. In some cases the evolved solutions exploited architectural complexities for even greater gains. Furthermore, the evolutionary system even performed well in randomly altered test environments. However, the genetic algorithm did not perform well in several situations. A discussion of those situations and suggestions for how to improve the evolutionary system follows.

5.2 Shortcomings

5.2.1 Analysis of Generated Distributions

It was initially hoped that the evolutionary system would generate cogent distributions that would instantly illuminate the best way to schedule tasks for the target architecture. The resulting algorithms could be inserted directly into a program at

compile time, avoiding the need for the lengthy evolutionary process. For relatively simple distributions onto simple architectures, the evolved algorithms are easy to understand. More complex work distributions evolved completely incomprehensible algorithms. By treating these algorithms as a black box and testing the outputs for a variety of inputs, it is possible to determine what the algorithm did. But even from this knowledge, determining how the algorithm works is difficult. Although these complexities do not lead to the discovery of new generalized algorithms, the results often do lead to interesting insights into the scheduling problem posed by a particular program and related target environment.

The evolved algorithms interact in very subtle ways. Some of the components are non-linear; the zero safe divide has separate behavior for a zero denominator, and the exponential function is capped at a maximum. The evolved algorithms would often utilize these irregularities to their advantage. Often an algorithm would have two very different behaviors over work distribution. The subtle non-linearities in the components permitted this behavior. By analyzing these break points in the evolved algorithms it is possible to better understand changes in the program's work distribution.

The evolutionary system can also expose subtle interactions that effect performance of the host architecture. Some algorithms seemed to distribute work in a random ordering, yet changing the ordering of iterations often proved harmful. Also, changing an offset for a periodic system changes the results. Operating under the standard assumptions of how the target architecture works, one would never suspect these changes to have any effect. Duplicating such performance gains in other target architectures, or even for different programs on the same target architecture is not possible. These gains are the result of some sort of inexplicable interaction between the architecture and the software and were not present in other systems.

Running multiple tests often helped identify interesting aspects of the test system. The best way to use the evolutionary system to better understand the performance characteristics of the target machine and software is to execute several trials. Multiple evolutionary experiments for the same program on the same architecture always

yielded seemingly different distributions. However, these different appearing expressions distributed the iterations in very similar ways. While many would be similar, equally as many would be completely different, yet still yield an optimal result. This finding tends to confirm earlier hypothesises about the nature of the search space. While the set of all possible expressions is large, the actual performance space is much smaller. Many expressions map to essentially the same work distribution, and a large number of the different work distributions are performance equivalent. By drawing on the similarities and differences from each of the trials, it is possible to better understand the tested system.

Though some of the tests produced results that seemed to find algorithms that may be more generally applicable, nearly always the evolutionary system found an easier way to achieve better performance by exploiting some program specific advantage. The tests that did find generally applicable results were for the most part contrived to extract a previously known result. The evolved solutions were only efficient for the specific evolved case and were too complicated to glean any knowledge of how to solve the scheduling problem in the general case. By examining the actual work distribution that arises from an algorithm, more could be learned about the optimized program and the target architecture but not about generalized algorithms.

5.2.2 Failures

While the evolutionary system proved to be very effective in many different test programs and target architectures, there are several common programming situations where the GA is not effective. These shortcomings arise due to a lack of parallelism in the test code and problems inherent to all genetic algorithms.

In order for the evolutionary system to be successful, the program must consist of a set of independent tasks that can be arbitrarily scheduled. The evolutionary system can only extract as much parallelism as is available from the test program. The target architecture can also limit the level of parallelism. For example, if all memory accesses are serialized on a single low bandwidth bus, then the total available parallelism is further reduced. The evolved solution will be optimal given the test environment,

but the result may seem poor given its relative performance compared to the serial version.

Also, the additional overhead of the inserted scheduling code can cause the evolved solution to perform significantly worse than the traditional techniques. The N-body simulation test highlighted this result by generating an optimal result that was worse than the serial running time of the program. In this case, the actual computation of the iteration assignment including the additional function call was more work than the body of the loop. The loop iterations must contain enough computation to hide the overhead of scheduling. Loops with a large number of low computation iterations can often be restructured to fewer iterations of more computation per iteration. In these situations, the GA can still be useful. In fact, this technique was used for the Monte Carlo simulation with much success.

The use of an genetic algorithm also restricts the class of potential programs. The evolutionary system must estimate the fitness of thousands of potential scheduling algorithms, so programs with long running times are not feasible. Also, programs with irregular performance characteristics are not ideal, since they must be executed many times in order to obtain an accurate fitness measurement. Since the genetic algorithm requires so many fitness evaluations, the test program execution is a limiting factor in the usefulness of the GA. Possible solutions to these problems are stated below.

5.3 Suggestions for Future Work

Many of the shortcomings of this research could be ameliorated by improving upon the system in several key areas. First, changing the evolutionary system could cause the genetic algorithm to evolve distributions that are easier to understand. The genome could be changed to allow self-defined functions [23]. This technique is an improvement upon genetic programming that allows evolved programs to define subroutines. Self-defined functions permit code reuse and encouraged a layered evolutionary procedure, much like the development of repeatable cell structures and organs in higher animals. This additional construct would yield programs that had definite structure,

instead of the single giant expressions the current system evolves. Another way to improve the clarity of the evolved algorithms would be to reward brevity in the fitness function. The fitness of an algorithm would be a function of both the running time and the total expression depth. Here the evolutionary system would work to balance both goals. Another option is to automatically reduce constant subexpressions in the algorithms during the evolutionary process [24]. This technique often leads to loss of genetic material, so another approach is to keep multiple populations, each ranked by a either a fitness metric of minimum time or minimum depth. This approach has been shown to achieve faster convergence without the loss of potentially useful genetic material [33]. Changes in the evolutionary system may lead to the evolution of more universally useful scheduling algorithms.

Another possible improvement would be to amortize the evolution time over the life of the program. Many applications require a long running time, which makes the evolutionary processes infeasible. One possible solution would be to build into the executable of a program an evolutionary system. Every time the program is executed, the evolutionary system would use the execution as a fitness evaluation. A capable stable of potential work distributions could be pre-evolved, eliminating disastrously inefficient distributions before the process begins. Pre-evolution would reduce the randomness in the system and may cause premature convergence, but it would eliminate exceptionally poor performance runs. This change would allow users to enjoy the gains of the evolutionary system without the computational expense.

Finally, the evolutionary system could be expanded to evolve many more parameters of performance oriented parallel programs. Parallel programs often need code restructuring and data distribution to perform optimally. These tasks are much like work distribution in that they are difficult to do by hand because the target architecture often has poorly understood performance characteristics. Also, these tasks would fit well within the framework of the symbiotic GA, since they have definite interdependencies. These additional components must be somehow represented as evolvable structures. With these additional elements, the evolutionary system should evolve better solutions to many more performance problems.

5.4 Closing Remarks

This research has shown that an evolutionary system can consistently generate optimized parallel programs that outperform traditional techniques. By varying the parameters of the genetic algorithm, work distributions for a wide variety of parallel programs and target architectures can be found. While the evolved work distributions may not perform well in the general case, they can be used to create better work distributions for similar programs on similar target architectures. Also, an analysis of the evolved distributions often reveals previously unknown characteristics of the target architecture. The evolutionary system is an efficient and automatic way to optimize parallel programs for a specific architecture without knowledge of the performance characteristics of the target system.

Bibliography

[1] Forman S. Acton. *Numerical Methods that Work.* Harper and Row Publishers. 1970.

[2] Anat Agarwal, et al. "The RAW Benchmark Suite: Computation Structures for General Purpose Computing" MIT Laboratory for Computer Science. Cambridge, MA. 1999.

[3] Arvind, et al. "Semantics of pH: A parallel dialect of Haskell." Haskell Workshop. 1995.

[4] David Bailry, et al. "The NAS Parallel Benchmarks 2.0" NASA Ames Research Center. Moffett Field, CA. 1995.

[5] Tobias Blickle and Lothar Thiele. "A Comparison of Selection Schemes used in Genetic Algorithms." Swiss Federal Institute of Technology. June 1995.

[6] Compaq MPI User Guide. Compaq Computer Corporation. Houston, TX. July 1999.

[7] DPF: A Data Parallel Fortran Benchmark Suite. [http://rodin.cs.uh.edu/ johnsson/dpf/root.html]. November, 1996.

[8] A. Drake. *Fundamentals of Applied Probability Theory.* McGraw Hill Publishers, New York. 1967.

[9] Lloyd D. Fosdick, Elizabeth R. Jessup, Carolyn J. C. Schauble, and Gitta Domik. *Introduction to High-Performance Scientific Computing.* The MIT Press. 1996.

[10] D. J. V. Evans, A. M. Goscinski. "Automatic identification of parallel units and synchronization points in programs for parallel execution on a distributed system" *Computer Systems Science & Engineering.* v 12 n 5 September 1997.

[11] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing. 1989.

[12] Goldberg, Kargupta, Horn, CantuPa. "Critical Deme Size For Serial and Parallel Genetic Algorithms." University of Illinois at Urbana-Champaign. January 1995.

[13] G. H. Golub and C. F. Van Loan. *Martrix Computations*, 2nd ed. Johns Hopkins University Press. 1989.

[14] A. Greenberg, J. Mesirov, and J. Sethian. "Programming Direct N-body Solvers on Connection" TMC technical report 245, August 1992.

[15] William Gropp. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press. 1999.

[16] J. H. Holland. "Adaption in Natural and Artificial Systems." University of Michigan, Department of Computer and Communication Sciences. 1975.

[17] J. H. Holland. "Genetic Algorithms and Adaption." (Technical Report No. 34) University of Michigan, Department of Computer and Communication Sciences. 1981.

[18] Peter T. Hraber, Terry Jones, and Stephanie Forrest. "The Ecology of Echo." *Artificial Life* Volume 3, Number 3. MIT Press. Summer 1997.

[19] Christoph W. Kebler. *Automatic Parallelization.* Lengerich, Germany. 1994.

[20] David E. Keyes. "Parallel Numerical Algorithms, An Introduction." *Parallel Numerical Algorithms*, 1-15. Kluwer Academic Publishers. 1997.

[21] Kuck & Associates, Inc. [http://www.kai.com]. November, 1999.

[22] John Koza. *Genetic Programming, on The Programming of Computers by Means of Natural Selection.* The MIT Press. 1994.

[23] John Koza. *Genetic Programming II.* The MIT Press. 1997.

[24] John Koza. *Genetic Programming III.* The MIT Press. 1999.

[25] MPI: A Message-Passing Interface Standard. [http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.html]. Febuary, 1997.

[26] MPICH-A Portable Implementation of MPI. [http://www-unix.mcs.anl.gov/mpi/mpich/]. December 2, 1999.

[27] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag. 1994.

[28] K. Bueno Muthukumar, F. Garcia de la Banda, M. Hermenegildo. "Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism." *Journal of Logic Programming.* v 38 n 2 Feb 1999. p 165-218

[29] OpenMP Fortran Application Program Interface. Specification version 1.0. October 1997.

[30] *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing.* Cambridge University Press, 1996.

[31] Richard B. Pelz. "Parallel FFTs." *Parallel Numerical Algorithms*, 245-266. Kluwer Academic Publishers. 1997.

[32] Martin C. Rinard, Daniel J. Scales, and Monica Lam. "Jade: A High-Level, Machine-Independent Language for Parallel Programming." *Computer.* June 1993.

[33] Conor Ryan. "Pygmies and Civil Servants." *Advances in Genetic Programming.* 1994.

[34] Conor Ryan. "GPRobots and GPTeams — Competition, co-evolution, and co-operation in Genetic Programming." 1994.

[35] Conor Ryan and Paul Walsh. "Paragen : A novel technique for the autoparallelisation of sequential programs using Genetic Programming." *The Proceedings of Genetic Programming.* 1996.

[36] Kamini Sharma. *A Parallel Simulated Annealing Algorithm for Module Placement in a Multi-Processor System.* Austrian Center for Parallel Computation, Vienna.

[37] Terence Soule and James A. Foster. "Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming." *Evolutionary Computation* Volume 6, Number 4. MIT Press. Winter 1998.

[38] "Stanford Parallel Applications for Shared-Memory" Stanford University, CA. 1994.

[39] Patrick H. Winston. *Artificial Intelligence* 3rd ed. Reading, Mass. Addison-Wesley Publishing Co. 1992.

[40] Joy Woller. "Basics of Monte Carlo Simulations." [http://wwitch.unl.edu/zeng/joy/mclab/mcintro.html]. Spring 1996.

6923-39