Efficient Utilization of Fine-Grained Parallelism using a microHeterogeneous Environment

by

William L. Scheidel

A Thesis Submitted

in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in

Computer Engineering

Primary Advisor: _____

Dr. Muhammad Shaaban, Assistant Professor

Committee Member: _____

Dr. Andreas Savakis, Associate Professor

Committee Member: _____

Dr. James Heliotis, Professor

Department of Computer Engineering

Kate Gleason College of Engineering

Rochester Institute of Technology

Rochester, New York

September 10, 2002

Release Permission Form

Rochester Institute of Technology

Efficient Utilization of Fine-Grained Parallelism using a microHeterogeneous Environment

I, William L. Scheidel, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce this thesis, in whole or in part, for non-commercial and non-profit purposes only.

William L. Scheidel

Date

ABSTRACT

Heterogeneous computing environments use an assortment of high performance machines with different architectures in an attempt to most efficiently execute the various tasks that are required by an application. While this environment is very well suited for tasks such as image understanding, heterogeneous processing has key limitations which including the lack of support for fine-grained parallelism, the high communication overhead of moving tasks between machines, and the prohibitively high costs.

The goal of this thesis is to propose a new computing paradigm, called micro-Heterogeneous computing or mHC, which incorporates PCI based processing elements (vector processors, digital signal processors, etc) into a general purpose machine. In this manner the benefits of heterogeneous computing on scientific applications can be achieved while avoiding some of the limitations. Overall performance is increased by exploiting fine-grained parallelism on the most efficient architecture available, while reducing the high communication overhead and costs of traditional heterogeneous environments. Furthermore, mHC based machines can be combined into a cluster, allowing both the coarse-grained and fine-grained parallelism to be fully exploited in order to achieve even greater levels of performance.

The ensuing chapters will provide the motivation for this work, an overview of heterogenous computing, and the details pertaining to microHeterogeneous computing. The framework implemented to demonstrate a microHeterogeneous computing environment will be examined as well as the results. Finally, the future of micro-Heterogeneous computing will be discussed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank everyone who supported or helped me in any way to complete this thesis:

- Dr. Muhammad Shaaban – For the idea of microHeterogenous Computing and allowing my to take part in it.
- Dr. Andreas Savakis – For providing help and counsel when needed.
- Dr. James Heliotis – For being a part of my committee.
- My family – For their support and understanding.

GLOSSARY

ANALYTICAL BENCHMARKING: Process of determining the suitability of a particular machine to execute a particular task

CPOP: Critical Path on a Processor Scheduling Heuristic

ETC: Estimated time to completion. The estimated amount of time that a particular task will require to execute on a particular device.

HEFT: Heterogeneous-Earliest-Finish-Time Scheduling Heuristic

MAPPING: Process of assigning a task to be executed on a particular machine

MHC: microHeterogeneous Computing

MIMD: Multiple Instruction, Multiple Data

MPI: Message Passing Interface

PVM: Parallel Virtual Machine

RC: Relative Cost Scheduling Heuristic

RTMM: Real-Time Min-Min Scheduling Heuristic

SIMD: Single Instruction, Multiple Data

SMM: Segmented Min-Min Scheduling Heuristic

VECTOR PROCESSING: Type of processing that operates on arrays of data elements simultaneously

WRTMM: Weighted Real-Time Min-Min Scheduling Heuristic

Chapter 1

INTRODUCTION

Heterogeneous computing environments use an assortment of high performance machines with different processing architectures in an attempt to most efficiently execute the various tasks required by an application. This type of environment has proven to be very effective for applications such as image understanding [27] where there are many different levels of processing that are best suited to various underlying architectures. However, there are limitations that prevent it from becoming applicable to an even wider set of problems.

One of the main drawbacks of heterogeneous environments is the granularity of the parallelism that can be supported. Due to the loosely coupled nature of the architecture, the grain size must be large enough to overcome the overhead required to send a task to a particular machine. This overhead is dependent on the size of the task, the working set required, and the type of interconnect used. Therefore, even if a machine might be able to provide significant performance gains for a particular task, the inherent overheads might prevent it from being efficiently used. Also, applications which consist of mostly finer-grained parallelism are unable to benefit from heterogeneous environments. Another hindrance to using heterogeneous environments is their cost effectiveness. There are very few applications that warrant the cost and time required to set up and maintain a heterogeneous computing environment.

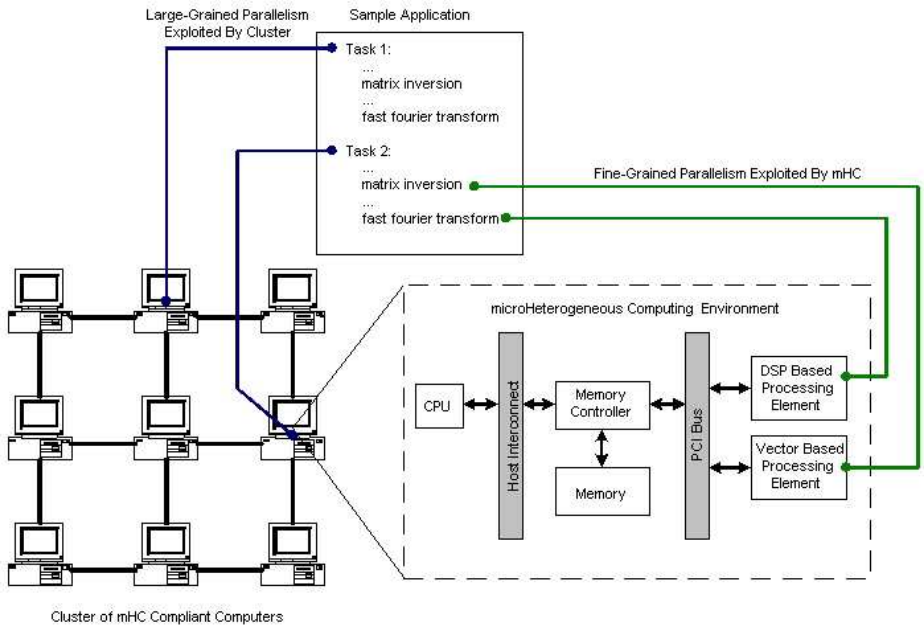

Figure 1.1: A Cluster of microHeterogeneous Computers

This thesis proposes a new computing paradigm, named microHeterogeneous computing or mHC, with the goal of achieving some of the benefits of a heterogeneous environment while avoiding the aforementioned limitations. The mHC environment incorporates PCI based processing elements (vector processors, digital signal processors, etc) into a general purpose computer thereby creating a small-scale heterogeneous system well suited to executing scientific applications. This environment is then able to exploit fine-grained parallelism by mapping individual function calls, defined by the mHC application programming interface, to the best suited processing element that is available. These function calls are then executed in parallel. Performance is increased by running tasks on the most compatible architecture, while no longer requiring both the high communication overhead and costs of traditional heterogeneous environments.

Furthermore, the mHC based machines can be combined into a cluster which allows both the coarse grained and fine grained parallelism to be fully exploited to achieve even higher levels of performance. This combination creates an extremely cost effective platform for utilizing various levels of parallelism and various architectures in order to achieve the highest performance. A sample mHC environment is depicted in Fig. 1.1 outlining these various levels of parallelism.

To demonstrate its effectiveness, the framework required to support mHC based applications on a single general purpose machine was implemented. This allowed for actual applications to be compiled and run using standard techniques. The framework consisted of an application programming interface based on a subset of the GNU Scientific Library [11], a real-time dynamic scheduling algorithm, and simulated devices that executed the various function calls. Since real applications were compiled and run under the framework, accurate performance numbers were obtained and clearly demonstrate the applicability of the mHC architecture.

The organization of the remainder of this document is as follows: Chapter 2 gives some background on heterogeneous computing, its limitations, and programming and scheduling methods; Chapter 3 is an overview of the microHeterogeneous computing environment; Chapter 4 discusses application programming interfaces and specifically ones relevant to scientific computing; Chapter 5 details the microHeterogeneous framework that was developed and how it was implemented; Chapter 6 presents the results obtained from various simulations using the framework developed. Finally, Chapter 7 presents the conclusions for this work, as well as ideas for future work.

Chapter 2

HETEROGENEOUS COMPUTING

The concept of microHeterogeneous computing is a variation on the standard heterogeneous computing environment. This chapter gives a brief background on heterogeneous computers including scheduling and programming techniques. The chapter concludes with a discussion on the limitations of heterogeneous environments.

2.1 Background

Heterogeneous computing is an architecture which provides an assortment of high performance machines for use by an application and arose from the realization that no single machine is capable of performing all tasks in an optimal manner. These machines differ in both speed as well as in capabilities and are connected using high speed, high bandwidth intelligent interconnects that handle the intercommunication between each of the machines. Heterogeneous computing is an extension to homogeneous computing, which uses only type of machine, that has the potential to increase performance and cost effectiveness.

The need for heterogeneous computers arose from the varying needs of today's most computation-intensive applications. These applications generally involve many different types of processing, such as SIMD, MIMD, and vector processing, which have the potential to be exploited. However, with traditional homogeneous computers only a single type of processing is able to be performed efficiently, while all others

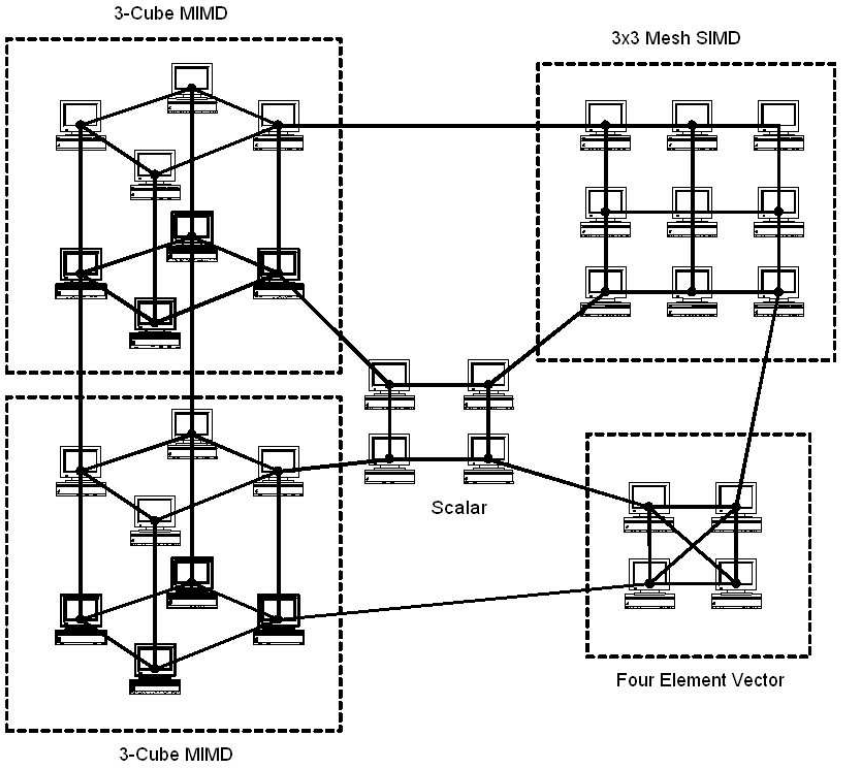

Figure 2.1: A Heterogeneous Environment

will suffer and thus limiting the effectiveness of these architectures. This is summarized by Amdahl's Law, which states that the performance of a system is dictated by the percentage of code that requires a type of processing not particularly supported by the hardware.

Heterogeneous computing tries to solve this problem by providing an assortment of machines that are capable of efficiently performing various tasks. Analytical benchmarking is used to determine the optimal speedup that a particular machine can achieve when it executes code that is best suited for that machine type. The relationship between this optimal speedup and the actual speedup achieved creates the

basis of determining how well a code segment is able to be matched to the machine [26]. The results of the analytical benchmarking are then used to determine feasible partitioning and mapping of applications.

Once all of the different machines have been benchmarked, the code to be executed must be profiled. Since Heterogeneous systems involve many different classes of high performance machines, it becomes essential to correctly identify the type of code that is to be run so that an attempt can be made to match it with the most efficient machine. Without this information, the performance enhancements that heterogeneous computering environments can provide are lost.

The profiling is done off-line and is used to determine the types of processing that exists in each program segment as well as the execution times. The different types that can be identified include: vectorizable decomposable, vectorizable non-decomposable, fine/coarse-grain parallel, SIMD/MIMD parallel, scalar, and special purpose [15]. An example of application profiling is shown in Fig. 2.2.

The program segments determined by the profiler must then be mapped onto the most efficient hardware in order to minimize the completion time of an application running on a heterogeneous computer. This efficiency depends heavily on computation costs, communication costs, and interference costs [21].

Computation costs involve the computation time of a particular task on a particular machine. Since a heterogeneous computer involves many different types of machines, the computation time of task is heavily dependant on the machine to which the task is assigned. The computation time is also dependant on the current load of

the assigned machine.

Communication costs involve the communication time between processors when tasks are divided across different machines and are dependent upon the type of interconnection network used and the bandwidth which is available. In order to realize the performance improvements offered by heterogeneous computing the communication costs must be minimized. The interconnection medium must be able to provide high bandwidth (multiple gigabits per second per link) at a low latency. It must also overcome current deficiencies such as the high overhead incurred during context switches, the overhead due to the need of executing high level protocols on each machine, and the overhead of managing large amounts of packets [15]. While the use of LANs has become commonplace, these types of connections are not well suited to heterogeneous supercomputers.

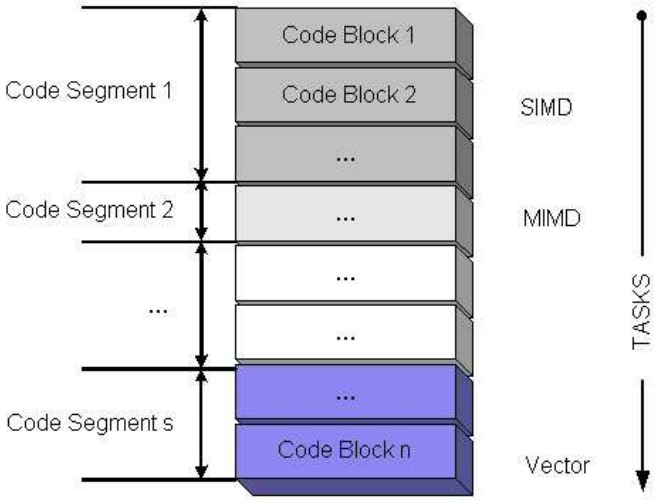

Figure 2.2: Code-Type Profiling Example

Interference costs are incurred when multiple tasks are assigned to a machine, which creates resource contention and reduces processor utilization. Interference costs increase overall completion time and therefore must also be minimized during the mapping.

Mapping a parallel program onto a parallel architecture to minimize completion time has already been shown to be an NP-hard problem even in a homogeneous environment [4]. The fact that a heterogeneous computer involves a myriad of machines simply complicates the matter further. The efficient mapping, or scheduling, of the tasks of an application on the available resources, however, is one of the key factors for achieving high performance and has become one of the most intensely studied issues dealing with heterogeneous computing environments.

2.2 Task Scheduling

Scheduling for a heterogeneous environment includes of all of the same issues found in scheduling for homogeneous environments plus some additional issues. Scheduling needs shared by both homogeneous and heterogeneous systems include job scheduling, intermediate-level scheduling, and low-level scheduling. Job scheduling involves the selection between available processes to run on the available hardware. The intermediate-level scheduling is in charge of smoothing operations over fluctuations in the current load of the system. The low-level scheduling determines the next process to run on a machine for a certain amount of time.

The scheduler in a heterogeneous environment must always be aware of the different task-types and machine-types in the system in order to schedule tasks to the

most appropriate machine as well as be prepared to reassign tasks in case the system configuration changes [15]. In addition, the scheduler must be acutely aware of the bottlenecks and queuing delays caused by the heterogeneity of the hardware.

The following are some of the more recently developed scheduling heuristics. Brief descriptions of other common mapping heuristics are also provided for completeness.

2.2.1 Segmented Min-Min

In the standard Min-Min scheduling heuristic, the minimum completion time for each task is computed with respect to each of the machines that are currently present. The task with the overall minimum completion time is selected and assigned to the corresponding machine. The task is then removed and the process continues until all tasks have been scheduled [2, 10]. This heuristic is very simple, fast, and provides good performance. One drawback, however, is that small tasks are assigned and executed first which leaves some machines idle while larger tasks are being executed. The Segmented Min-Min heuristic proposed in [29] attempts to solve this problem by altering the Min-Min algorithm to schedule large tasks first.

The Segmented Min-Min algorithm first computes the expected time to compute (ETC) for each task on each machine producing an ETC matrix where ETC(i,j) is the ETC for task i on machine j. The tasks are then sorted into a task list in descending order which has the effect of promoting tasks with large ETC values. The task list is divided into n segments and for each segment, Min-Min is applied to assign tasks to machines. The full algorithm is shown below.

Segmented min-min (Smm)

1. Compute the sorting key for each task:

   Sub-Policy 1 - *Smm-avg*: Compute the average value of each row in ETC matrix

   $$key_i = \sum_j \frac{ETC(i,j)}{m}$$

   Sub-Policy 2 - *Smm-min*: Compute the minimum value of each row in ETC matrix

   $$key_i = \min_j ETC(i,j)$$

   Sub-Policy 3 - *Smm-max*: Compute the maximum value of each row in ETC matrix

   $$key_i = \max_j ETC(i,j)$$

2. Sort the tasks into a task list in decreasing order of their keys.
3. Partition the tasks evenly into N segments.
4. Schedule each segment in order by applying Min-min.

Even though Segmented Min-Min only proposes a slight change to the standard Min-Min algorithm it successfully improves the load balancing and enhances the performance from 2% to 12%. The scheduling time is also reduced because the Segmented Min-Min uses a divide and conquer strategy which reduces the search space of the Min-Min algorithm when determining which task to map [29].

2.2.2 Relative Cost Algorithm

The Relative Cost Algorithm uses a relative cost criterion rather than prioritizing tasks based on size. The relative cost of a task is calculated by dividing the task completion time $ct_k(i,j)$ by the average completion time of tasks i. This assures that a higher priority is given to tasks that have a good match between tasks and machines and minimizes the overall completion time [28]. The complete Relative Cost Algorithm is shown below.

RC Algorithm

1. For each task i and machine j, let $ct(i,j) = ETC(i,j)$ and $\rho(i) = 0$

$$\gamma_s(i,j) = ETC(i,j)/ETC(i)_{avg}$$

2. For $k = 1$ to t do

   (a) for task i, $1 \leq i \leq t$ and $\rho(i) = 0$

      i. compute $ct_k(i)_{avg} = \sum_{i \leq j \leq m} ct_k(i,j)/m$

      ii. select machine B_i such that $ct_k(i,B_i) = \min_{i \leq j \leq m} ct_k(i,j)$

      iii. compute $\gamma_d(i,B_i) = ct_k(i,B_i)/ct_k(i)_{avg}$

   (b) select task A_k such that

$$\gamma_s(A_k,B_{A_k})^\alpha \times \gamma_d(A_k,B_{A_k}) = \min_{1 \leq i \leq t, \rho(i)=0} \gamma_s(i,B_i)^\alpha \times \gamma_d(i,B_i)$$

   (c) let $\rho(A_k) = 1$ and $F(A_k) = B_{A_k}$, that is, assign task A_k to machine B_{A_k}

   (d) for task i with $1 \leq i \leq t$ and $\rho(i) = 0$, modify

$$ct_{k+1}(i,j) = \begin{cases} ct_k(i,j) & \text{if} j \neq B_{A_k} \\ ct_k(i,j) + ETC(A_k,B_{A_k}) & \text{otherwise} \end{cases}$$

The factor of $\gamma_d^\alpha \times \gamma_s$ is used to fine tune the matching and load balancing criteria. The smaller the γ_d the better the matching of machines. Therefore tasks that match machines better will be given a higher priority. The α parameter is used to adjust γ_s, or the static relative cost, and is always between $0 \leq \alpha \leq 1$. The specific value for α is generally determined by way of experimentation.

In practice, the Relative Cost Algorithm provides a good compromise between load balancing and matching proximity and consistently performs better then Min-Min, Max-Min and GA.

2.2.3 Heterogeneous-Earliest-Finish-Time and Critical Path on a Processor

Instead of using a strict performance measurement such as estimated completion time or relative cost, Heterogeneous-Ealiest-Finish-Time (HEFT) and Critical Path on a Processor (CPOP) determine task scheduling based on the application's task graph [23]. The task graph is a directed acyclic graph, denoted by $G = (V, E)$, where V is the set 0f v tasks and E represents the set of e edges between the tasks. The edges represent the dependencies between the tasks, with edge (i, j) meaning that task n_i must be completed before task n_j may begin. Two important properties of the task graph are the entry and exit tasks. Entry tasks are those tasks that have no parent task and exit tasks are those tasks that have no children. A cost matrix, W, is calculated in which each $w_{i,j}$ represents the estimated execution time of task n_i on processor p_j.

Tasks in this system are ordered by their priorities which are linked to their upward and downward ranking. The upward rank is calculated recursively by traversing the graph upward and is defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \tag{2.1}$$

where $succ(n_i)$ is the set of successors of task n_i, $\overline{c_{i,j}}$ is the communication cost of edge (i, j), and w_i is the average computation cost for task n_i. Therefore, the resulting $rank_u(n_i)$ defines the length of the critical path from task n_i to the exit task and includes the computation cost for task n_i. The downward rank is very similar but is

calculated by traversing the graph upward and is defined by

$$rank_d(n_i) = \overline{w_i} + \max_{n_j \in pred(n_i)} (\overline{w_j} + \overline{c_{j,i}} + rank_d(n_j)) \tag{2.2}$$

where $pred(n_i)$ is the set of predecessors of task n_i. The resulting $rank_d(n_i)$ is the longest distance from the entry task to the task n_i but does not include the computation cost of task n_i.

The HEFT algorithm consists of two phases, a prioritizing phase and a processor selection phase. During the prioritizing phase, the task list is sorted by decreasing order of $rank_u$. This provides a linear ordering of the tasks which preserves the precedence constraints.

Once the tasks have been properly ordered, the tasks are mapped to the processors during the processor selection phase. During this phase, an appropriate idle time slot is located for each task. The search starts at the time equal to the $ready\_time$ of n_i on p_j, or in other words the time when all input data of task n_i has arrived at processor p_j. The search continues until finding the first idle time slot that is capable of holding the computation cost of task n_i. HEFT differs from most mapping algorithms because it considers assigning tasks in idle time occurring between two previously scheduled tasks instead of automatically beginning the search for an appropriate processor starting at the time the last scheduled task has completed. The complete HEFT algorithm is shown below.

HEFT Algorithm

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ of tasks by traversing graph upward, starting from the exit task.
3. Sort the tasks in a scheduling list by nonincreasing order of $rank_u$ values.
4. while there are unscheduled tasks in the list do
   (a) Select the first task, n_i from the list for scheduling.
   (b) for each processor p_k in the processor-set ($p_k \in Q$) do
      i. Compute $EFT(n_i, p_k)$ value using the insertion-based scheduling policy.
   (c) Assign task n_i to the processor p_j that minimizes EFT of task n_i.
5. endwhile

The CPOP algorithm consists of same two main phases as the HEFT algorithm, a prioritizing phase and a processor selection phase. During the prioritizing phase the upward and downward ranks of all of the tasks are calculated. The priority of the tasks is the summation of the upward and downward ranks. The critical path of the application is then determined. First the entry task is selected and marked as a task of the critical path. The successor with the highest priority is then selected and marked as part of the critical path. This continues until the exit task is reached.

Once the prioritizing phase is complete, the tasks are then mapped to processors in the heterogeneous system. Tasks that are part of the critical path are mapped to the critical-path processor, p_{cp}, which minimizes the cumulative computation costs of the tasks on the critical path. Tasks that are not part of the critical path are assigned to a processor which minimizes the earliest execution finish time of the task. The complete CPOP algorithm is shown below.

CPOP Algorithm

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ of tasks by traversing graph upward, starting from the exit task.
3. Compute $rank_d$ of tasks by traversing graph downward, starting from the entry task.
4. Compute $priority(n_i) = rank_d(n_i) + rank_u(n_i)$ for each task n_i in the graph.
5. $|CP| = priority(n_{entry})$, where n_{entry} is the *entry* task.
6. $SET_{CP} = priority(n_{entry})$, where SET_{CP} is the set of tasks on the critical path.
7. while n_k is not the exit task do
   (a) Select n_j where $((n_j \in succ(n_k))$ and $(priority(n_j) == |CP|))$.
   (b) $SET_{CP} = SET_{CP} \cup n_j$.
   (c) $n_k \leftarrow n_j$
8. endwhile
9. Select the critical path processor (p_{CP}) which minimizes $\sum_{n_i \in SET_{CP}} w_{i,j}, \forall p_j \in Q$
10. Initialize the priority queue with the entry task.
11. while there is an unscheduled task in the priority queue do
    (a) Select the highest priority task n_i from priority queue.
    (b) if $n_i \in SET_{CP}$ then
        i. Assign the task n_i on p_{CP}
    (c) else
        i. Assign the task n_i to the processor p_j which minimizes the $EFT(u_i, p_j)$.
    (d) Update the priority queue with the successors of n_i, if they become ready tasks.
12. endwhile

When tested on 56,000 randomly generated task graphs, HEFT and CPOP on average performed better then other list scheduling heuristics such as Dynamic-Level Scheduling [20], Mapping Heuristic [9], and Levelized-Min Time [14]. HEFT produced schedules that reduced execution time in 86% of the fifty-six thousand randomly generated task graphs, while CPOP produced better schedules in 65% of the cases. Due to the promising results and fast scheduling times, HEFT is currently being further developed [23].

2.2.4 Other Mapping Heursitics

Many other mapping heurisitics have been designed to map tasks onto a heterogeneous environment of which only a sample are described here. All of the following heuristics statically schedule meta-tasks (a set of tasks with no dependencies) onto the machines. This requires that both the number of tasks and the number of machines is known before the scheduling process begins. All of these heuristics have been evaluated in [5].

OLB: Opportunistic Load Balancing assigns each task in arbitrary order to the first available machine [2]. This technique generally does not produce acceptable schedules.

UDA: User-Directed Assignment assigns each task to the machine that has the best expected execution time [2]. This technique generally does not produce acceptable schedules.

Fast Greedy: Assigns each task to the machine with the minimum completion time [2].

Max-Min: A variation to Min-Min in which the task with the overall maximum completion time from the set of all unmapped tasks is selected and assigned to the corresponding machine [2, 10]. This technique generally does not produce acceptable schedules.

GA: The genetic algorithm operates on a population of chromosomes for a given problem and is used to search a large solution space. The initial population is generated randomly, though it could be generated using one of the other

heuristics available [25]. This algorithm generally provides for good performance though it is much slower then Min-Min while only performing slightly better.

A*: A* uses a tree based method in order to incrementally build a solution for the task mapping. The root node of the tree is generally that of a null solution, intermediate nodes represent partial solutions, and leaf nodes represent final solutions. The nodes are rated by a cost function and the node with the minimum cost function is replaced by its children while the node with the highest cost function is removed [7]. This algorithm provides very good schedules for some situations but is very slow, averaging about 1200 times slower then that of Min-Min [5].

2.3 Programming

Parallel programming environments include the tools needed to write and debug applications for parallel machines. These include programming languages, compilers, debuggers and other aides. A few of the different programming languages currently available are discussed here.

2.3.1 Parallel Virtural Machines (PVM)

PVM (Parallel Virtual Machine) is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource [18]. The overall objective of the PVM system is to to enable such a collection of computers to be used cooperatively for concurrent or parallel computation.

The unit of parallelism in PVM are tasks which are generally Unix processes. These tasks are independent sequential threads of control that alternate between

communication and computation. The system allows users to specify the set of machines for an application to use. The user also has the choice to view these resources as an attributeless collection of virtual processing elements or choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers

A system level process runs on each of the computer systems that permits the collection of machines to be viewed as a coherent system. The system supports process management, communication via message passing, and synchronization through the use of named barriers. Important to heterogeneous computing, PVM allows multifaceted virtual machines to be configured within the same framework and permits messages containing more than one datatype to be exchanged between machines having different data representations. Examples of programs that have been executed under PVM include matrix factorization, stochastic simulation of toroid networks, and Mandlebrot image computations.

2.3.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [1] is another programming environment used for parallel processing and is compatible with heterogeneous environments. The goal of MPI is to create a widely used standard for writing message passing based applications. The interface provided is practical, portable, efficient, and flexible and has become widely used.

The implementation of MPI is fundamentally different then PVM. MPI does not explicitly create processes to divide up among the various processing elements. In-

stead each processor is assigned a rank which then determines which parts of the application that processor will execute. This division of tasks among processors is left solely up to the developer writing the application.

The MPI standard includes routines to do point-to-point communication between two processing elements, collective operations to simultaneously communicate information between all processing elements, and implicit as well as explicit synchronization. The standard does not provide direct support for shared memory operations or support for threads. MPI has most recently been used by MPEG encoding and decoding applications [3].

2.4 Limitations

Although heterogeneous computing environments have proven to be very useful for some applications, it has drawbacks which prevent its use from becoming widespread. These limitations include the focus on coarse-grain parallelism, the inherent communication costs, and the actual implementation costs.

Heterogeneous computing environments function most efficiently on applications that demonstrate a very coarse-grained parallelism. This coarse-grained parallelism results in large task sizes and reduced coupling which allows the processing elements to work more efficiently. This requirement is also translated to most heterogeneous schedulers since they are based on the scheduling of meta-tasks, i.e. tasks that have no dependencies.

While some applications such as image understanding [27] and gaussian elimina-

tion [23] contain this type of parallelism, there are many applications that do not. These types of applications may have finer-grained parallelism or tend to be more tightly-coupled and are not able to benefit as much from a standard heterogeneous environment. In these cases, the task size is smaller and the overhead required to distribute the tasks becomes greater then the performance enhancement achieved by executing the task on a different architecture. This is also true with applications that include many dependencies which require more data to be passed between machines.

The inherent communication costs are another drawback of using a heterogeneous environment. Generally, specialized interconnects must be used in order to provide a high bandwidth low latency connection [15]. These specialized interconnects tend to be very expensive and are not always available. The other option is to use standard networking equipment (LANs) which is more cost effective but can greatly hinder the overall performance of the heterogeneous environment. As the granularity of an application decreases, the performance characteristics of the interconnect must increase or risk becoming a bottleneck in the system.

Finally, the implementation cost of heterogeneous networks is very prohibitive. Creating a heterogeneous network requires specialized machines and high speed interconnects in order to produce an environment with good performance characteristics. The cost might be acceptable for applications that are known to map well onto this type of environment, but in other cases they are simply not cost effective.

The next chapter discusses the role of application program interfaces and specifically describes a few scientific APIs that play a role in microHeterogeneous computing.

Chapter 3

APPLICATION PROGRAM INTERFACES

3.1 Background

An application program interface (API), sometimes referred to as an application programming interface, is the interface by which an application program accesses services provided by an operating system or other applications. An API provides a level of abstraction between the application and functionality that the API provides to ensure the portability of the code.

The use of APIs in modern application development is omnipresent. Computers have become so complex that low-level functionality is constantly being abstracted to higher levels through the use of these interfaces. Operating systems are required to provide one of the most diverse APIs that becomes the bridge between user applications and the hardware upon which they are executing.

Using an API to provide access to underlying functionality has some important benefits. First it allows an application to maintain portability. As long as the interface provided does not change, an application using the API is not disrupted if the functionality that the API provides is modified. This is especially true of APIs that provide an interface to libraries that are often optimized for different architectures.

The use of APIs also makes software development easier and more robust. Software developers do not need to concern themselves with low-level details such as drawing pixels on a screen, since these types of things have already been abstracted to high level APIs such as OpenGL [19]. APIs that have become a standard are also widely used and therefore widely tested, making the functionality that they provide less prone to errors and more dependable.

For microHeterogeneous computing environments, the most important APIs are those dealing with scientific computing.

3.2 Scientific Computing APIs

Heterogeneous and microHeterogeneous computing are mainly focused on scientific based applications since it is these types of programs that can benefit most from heterogeneous environments. These APIs provide access to the basic building blocks of scientific computing which can be utilized to create complete applications. A few of the more common scientific computing APIs are discussed here.

3.2.1 Basic Linear Algebra Subprograms (BLAS)

The Basic Linear Algebra Subprograms (BLAS) [17] are a set of low level operations that are fundamental to numerical linear algebra. The motivation was to create a highly optimized set of routines common to most scientific applications in order to improve the overall performance. Since a significant amount of execution time is generally spent in these low level operations, reducing the time required to perform these low level operations leads to an overall reduction in an applications execution time.

BLAS is divided into three separate levels dependent on the type of data that is operated on. The Level 1 BLAS, implemented between 1973 and 1977, includes subprograms for scalar and vector operations. The software package LINPACK uses the Level 1 BLAS extensively for the solution of dense and banded linear equations and linear least squares problems. The Level 2 BLAS, implemented between 1984 and 1986, deals with matrix-vector operations and the Level 3 BLAS, implemented between 1987 and 1988, deals with matrix-matrix operations. The linear algebra software package LAPACK utilizes the Level 2 and 3 BLAS for portable performance. Overall, the BLAS have enabled many applications to improve performance while maintaining portability.

Orginally, the BLAS was implemented in Fortran 66 but specifications for Fortran 77, Fortran 95, and C now exist. Highly efficient machine-specific implementations of the BLAS are available for almost every modern computer architecture. In general, the BLAS has been very well received by developers for the high performance of the library routines and the portability that comes with using a standardized API. The routines provided have become the building blocks of a large portion of scientific applications over the past two decades.

3.2.2 Vector, Signal, and Image Processing Library (VSIPL)

The Vector, Signal, and Image Processing Library (VSIPL) [13] was designed to provide a portable, object-based API for signal and image processing on embedded systems. The API began development in 1996 by Hughes Research Laboratory and was funded by DARPA. The working group included companies such as Lockheed-

Sanders, Nothrup Grumman, Digital, Intel, and Cray with the common goal to create an industry supported standard for vector and signal processing primatives.

One of the technical goals that VSIPL set to achieve was to be very portable. To accomplish this, the VSIPL API is implemented in ANSI C and requires only a simple re-compile to port between platforms. The API also does not restrict the actual implementation of the computational portion of the standard. This allows vendors to create optimized implementations, if desired, while maintaining source-code portability based on the API.

VSIPL is an object-based library, which is a departure from traditional libraries, such as BLAS, that are functional based. Applications utilizing the VSIPL API use special abstract data types whose implementations are hidden to allow for vendor-private implementations. The main data types used are *blocks* and *views*. *Blocks* are contiguous storage areas in which the actual data values are stored. *Views* are then created to determine how a *block* of data is handled. The three main *view* characteristics are offset from the beginning of a *block*, the number of elements (length), and the spacing between the elements (stride).

For example, a group of nine data elements would be stored in a contiguous storage area within a *block* as shown in Table 3.1(a). In order to create a *view* for the entire *block* as a vector, the offset would be set zero, the length would be set to nine, and the stride would be set to one. If only the even elements were desired, a *view* would be created with an offset of one, a length of four, and a stride of two. This *view* would only contain the data elements shown in Table 3.1(b). The *block* could

1	2	3	4	5	6	7	8	9

(a) A block of data

2	4	6	8

(b) A view of even elements

1	2	3
4	5	6
7	8	9

(c) A matrix view

Table 3.1: Example of VSIPL Blocks and Views

also be used as a 3×3 matrix by creating a *view* with an offset of zero, a row length of three, a row stride of one, a column length of three, and a column stride of three. This *view* would contain all of the data elements but would be used like a matrix as shown in Table 3.1(c).

The VSIPL API supports a wide range of functions that are most common to scientific based applications with an emphasis on signal and image processing. A summary of the functions included is located in Table 3.2.

VSIPL also includes special accommodations for embedded applications. Better performance is achieved through the use of early binding which allocates resources for an operation as early as possible. The memory space is also divided into a user data space and a VSIPL data space in order to assure that data is not corrupted. The user data space contains data the user is able to manipulate directly. Only data in the VSIPL data space can operated on by VSIPL operations and is hidden from direct

Signal Processing	**Vector/Matrix Ops**	**Linear Algebra**
▷ FFT ▷ Convolution ▷ Correlation ▷ FIR/IIR Filters	▷ Arithmetic ▷ Comparison ▷ Selection Operations ▷ Boolean Operations ▷ Data Conversion	▷ Inner, Outer, Kronecker Product ▷ Matrix-Vector, Matrix-Matrix Multiply ▷ QR, LU, Cholesky Decompositions ▷ Solvers

Table 3.2: Summary of VSIPL Functionality

user manipulation. In order to make development on an embedded device easier, VSIPL includes special development modes with higher degrees of error checking and a production mode which runs faster and includes no error checking to save space in embedded devices.

3.2.3 GNU Scientific Library (GSL)

The GNU Scientific Library (GSL) [11] is a modern numerical library for C and C++ programmers. The project was started in 1996 at the Los Alamos National Laboratory by Dr M. Galassi and Dr J. Theiler after they became discouraged with the restrictions of the licenses of other existing libraries. With this in mind, the GSL was released under the GNU General Public License (GPL) [12], making the library freely available.

With support for over 1000 functions, the GSL aims to provide the most comprehensive scientific computing library available. A summary of the topics covered are shown in Table 3.3. The GSL has been successfully compiled under all of the modern operating systems including SunOS, Solaris, Linux, HP-UX, IRIX, BSD, Windows, and Apple Darwin.

The GSL successfully combines a very powerful library with an easy to use and understand interface. The library uses an object-oriented design and allows different algorithms to be easily plugged in or changed at run-time without the need to recompile the actual application. The function calls follow a standard naming convention and data types which make the library particularly easy for the ordinary scientific user. It is also thread-safe which is critical for multi-threaded applications.

Complex Numbers	Roots of Polynomials	Special Functions
Vectors and Matrices	Permutations	Sorting
BLAS Support	Linear Algebra	Eigensystems
Fast Fourier Transforms	Quadrature	Random Numbers
Quasi-Random Sequences	Random Distributions	Statistics
Histograms	N-Tuples	Monte Carlo Integration
Simulated Annealing	Differential Equations	Interpolation
Numerical Differentiation	Chebyshev Approximation	Series Acceleration
Discrete Hankel Transforms	Root-Finding	Minimization
Least-Squares Fitting	Physical Constants	IEEE Floating-Point

Table 3.3: Summary of GSL Functionality

Another benefit to the GSL is its compatibility with other scientific based APIs. It provides direct support for all three levels of the BLAS, making the transition from the BLAS to the GSL a trivial matter. The GSL also uses the same *block* and *view* representations of data that VSIPL uses. This allows the GSL function calls to seamlessly interact with embedded devices based on the VSIPL standard. It is because of this compatibility with both the BLAS and VSIPL, and the comprehensive library, that the microHeterogeneous computing API is based on a subset of the GSL.

The next chapter discusses the microHeterogeneous architecture that is being proposed. It includes the benefits of heterogeneous computers while removing some of the inherent limitations.

Chapter 4

MICROHETEROGENEOUS COMPUTING

This chapter gives an overview of microHeterogeneous computing.

4.1 Description

The microHeterogeneous computing (mHC) environment is a new computing paradigm that attempts to most efficiently exploit the fine-grained parallelism found in most scientific computing applications. The environment is contained within a workstation and consists of a host processor and a number of additional PCI based processing elements as shown in Fig. 4.1. These processing elements might be DSP based, vector based, FGPA based, or even reconfigurable computing elements. In combination with a host processor, these elements create a small scale heterogeneous computing environment that has many of the same benefits of standard heterogeneous environments while also including the additional benefits of a shared-memory system with low overheads.

The idea for mHC environments originated with the observation that workstations are already becoming heterogeneous in nature. Graphics cards now come equipped with their own graphics processors, sound cards include digital signal processing hardware, and network cards contain special purpose network processors. However, these processing elements are not available to general user programs. There is simply no

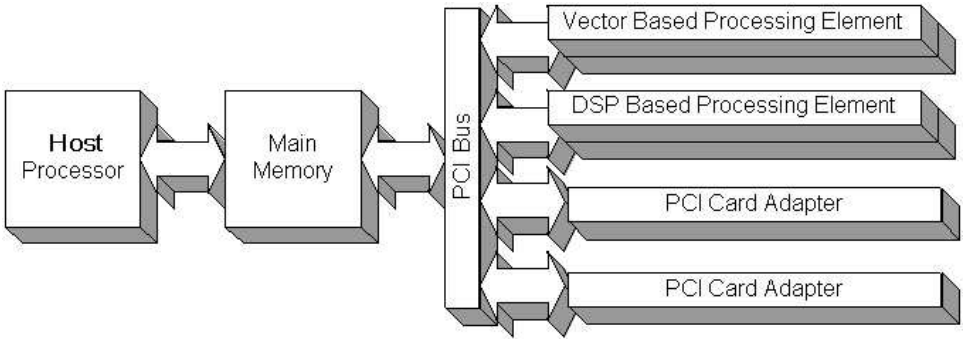

Figure 4.1: A microHeterogeneous Environment

way to compute fast fourier transforms on the DSP processor located on a sound card for example. This extra processing power, therefore, often goes unused.

There are, however, other PCI-based processing elements that are available to user programs, two of which are discussed later in this chapter. These add-on processors are able to execute some tasks much faster than a standard processor. One of their main drawbacks, however, is their cumbersome nature. This is why the use of such processing elements is not more wide spread. There is currently no standard API that manufacturers currently use, instead multiple proprietary APIs are in use depending on the type of device. This makes working with one of these devices difficult and working with combinations of these devices quite a challenge. In addtion, the developer is left to deal with the difficult issues of task mapping and load balancing, issues with which they may not be intimately familiar.

The proposed microHeterogeneous computing environment greatly simplifies the use of these types of devices for scientific computing. The mHC API is used to make function calls that become tasks. After a task has been scheduled, the call

immediately returns which allows for the user application to continue execution. In this manner, multiple tasks can be scheduled and executed in parallel even if the user application was not initially multi-threaded. It becomes possible for a developer to write an application once using the mHC API and then simply add more processing power, at a later time, if it becomes required.

4.2 mHC API

A standard API was developed that all mHC compliant devices will use, eliminating the need to learn a different interface for each device. When an API call is made, the framework handles all of the scheduling and load balancing issues internally by placing the task in the task queue of the most appropriate device. Only the functions available in the API can be scheduled to processing elements in the microHeterogeneous environment.

Scientific APIs take years to create and develop and more importantly, be adopted for use. We therefore decided to create an extension on an existing scientific API and add compatibility for mHC environment rather then developing a completely new API. This approach has many benefits. First, The mHC API can be more complete than would have been possible if a completely new API had been created. Secondly, it can leverage the numerous data types already available in the existing API. Lastly, building off of an existing API means that there is already a user base developing applications that would then become suitable for mHC.

After reviewing a large number of scientific APIs, it was decided that the GNU Scientific Library (GSL) would form the basis of the mHC API. The GSL is an ex-

Vector Operations	Matrix Operations	Polynomial Solvers
Permutations	Combinations	Sorting
Linear Algebra	Eigenvectors and Eigenvalues	Fast Fourier Transforms
Numerical Integration	Statistics	

Table 4.1: Scientific Areas Supported by the mHC API

tensive, free scientific library that uses a clean and straightforward API. It provides support for the Basic Linear Algebra Subprograms (BLAS) which is widely used in applications today and is data-type compatible with the Vector, Signal, and Image Processing Library (VSIPL) which is becoming one of the standard libraries in the embedded world.

Currently, the mHC extension to the GSL includes over sixty functions. The different areas of support are shown in Table 4.1. A full listing of the entire API can be found in Appendix A.

4.3 Example Devices

Currently there are no mHC compliant devices, but there are several PCI-based accelerator cards that could be utilized by an mHC environment. The only change required would be to update the device drivers in order to utilize the mHC API instead of the proprietary APIs that are currently implemented. A sample DSP accelerator card, the XP-15, and a sample vector processing accelerator card, the Pegasus-2, are described here. These devices also served as the model devices that were used to obtain the results located in Chapter 6.

Figure 4.2: The XP-15 DSP Accelerator Card

4.3.1 XP-15

The XP-15 [22] is a powerful DSP accelerator card developed by Texas Memory Systems. The card interfaces with a standard 64-bit, 66 MHz PCI bus making it fully compatible with the requirements of microHeterogeneous computing. The XP-15 is based around the TM-44 DSP Blackbird chip which is able to perform eighty 32-bit floating-point operations per instruction cycle. This results in a peak processing rate of eight GFLOPS. The addition of this card to a workstation results in a performance gain of 5x to 20x when dealing with DSP related functions. A comparison of the XP-15 to a 1.4 GHz Intel P4 for computing complex fast fourier transforms is shown in Table 4.2.

The XP-15 includes 256 megabytes of fast local Double Data Rate (DDR) memory directly connected to the TM-44 chip by way of a 256-bit DDR bus that provides 6400 MB/sec memory bandwidth. Data can be transferred into this local memory by means of the PCI bus in parallel with the TM-44 executing instructions. This is critical for the efficient processing of data. The host processor is also free to execute

CFFT Size	XP-15(microseconds)	1.4GHz P4(microseconds)	Improvement
1K	8	175	22x
4K	31	825	26x
16K	162	4,500	27x
64K	655	23,750	36x
256K	3,275	116,750	36x
1024K	13,107	550,000	42x

Table 4.2: Comparison of the XP-15 and a 1.4 GHz Intel P4 [22]

other tasks while the XP-15 is working on the data stored in its large local memory.

The XP-15 is capable of executing over 500 different scientific algorithms which have all been optimized to the hardware. These algorithms include real and complex vector arithmetic, real and complex matrix arithmetic, 1-D and 2D FFTs, image processing algorithms, and digital signal processing algorithms. Almost all of this functionality is mirrored in the mHC API making this device a particularly good match to microHeterogeneous computing.

4.3.2 Pegasus-2

The Pegasus-2 [6] is a high-performance digital signal processing board distributed by Catalina Research based on the Pathfinder-1 FFT vector processor. The board interfaces with a standard 64-bit, 66 MHz PCI bus making it fully compatible with the requirements of microHeterogeneous computing.

Processing on the Pegasus-2 is accomplished by passing data from one of the two input memories, through the Pathfinder-1, and into the output memory. The data is then passed back into one of the input banks if additional processing is required.

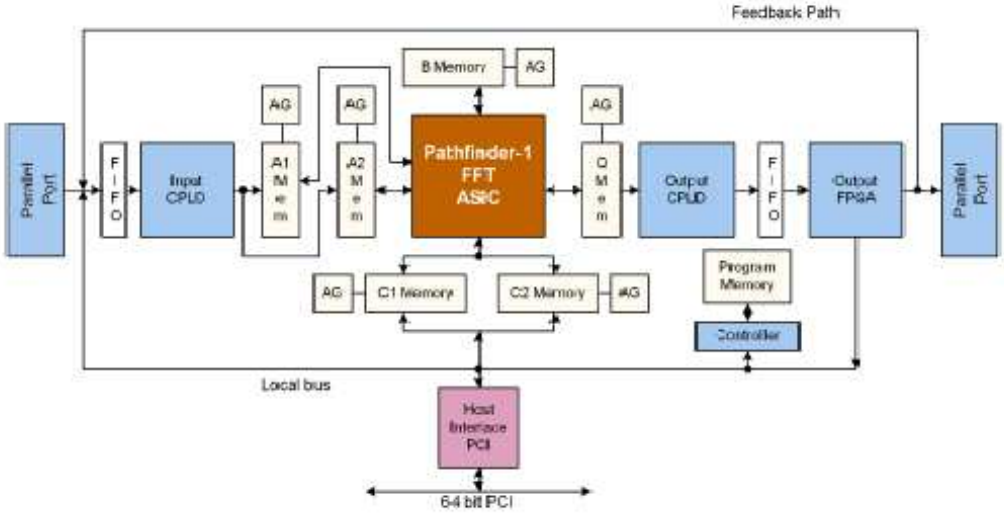

Figure 4.3: The Pegasus-2 Vector Processing Accelerator Card [6]

Each of the SRAM memory banks is 48 bits wide and 256K words deep.

The Pegasus-2, along with its multiple Sojourner-2 address generators, supports standard radix FFT patterns, real FFT operations, offset and stride addressing, and stacked transform operations. This allows the Pegasus-2 to handle very high speed complex vector multiplications, fast convolutions, polyphase filters, and 2D FFTs. Also included is a high performance reconfigurable computing element that follows the vector processor in the data path. This reconfigurable computing element can be configured on the fly by way of the PCI bus and has been used to support things such as FIRs and adaptive filtering. Some sample performance numbers can be found in Table 4.3.

Operation	Processing Time
1K complex FFT	25.6 microseconds
64K	3.28 milliseconds
256K	13.11 milliseconds
4096 tap complex FIR	1,235 microseconds
4:1 Polyphase filter	75.48 microseconds

Table 4.3: Pegasus-2 Sample Operation Performance [6]

4.4 Comparison to Heterogeneous Computing

As mHC is a derivative on heterogeneous computing, many similarities exist between the two. There are also some key differences which significantly change the role of mHC environments.

4.4.1 Task Granularity

As with heterogeneous environments, the need for mHC environments stems from the fact that applications generally involve many different types of processing and parallelism which have the potential to be exploited. In order to handle the different processing needs of an application, both environments provide an assortment of processing architectures so that tasks may be executed on the one which is most suitable. However, while standard heterogeneous environments may contain different types of machines to efficiently execute tasks, only coarse-grained parallelism is supported. The microHeterogeneous computing environment instead focuses on the fine-grained parallelism by providing the processing elements within a single machine which creates a tightly-coupled shared-memory environment that is well suited to this application. Task size is reduced to a single function call, such as matrix inversion or a fast fourier transform, instead of an entire procedure that may contain many

instructions and function calls.

There are some obvious drawbacks to this technique. First, PCI based processing elements are not nearly as powerful as the machines used in a standard heterogeneous environment. Second, there is a small and finite number of processing elements that can be added to a single machine. However, neither of these are an issue when dealing with parallelism at such a fine level. While the PCI based processing elements may not provide as much of a speedup on each task they execute, the shear number of tasks being executed makes up for this. Over the execution time of an application, the performance gains accumulate and positively effect the overall execution time. Additionally, since the task size is so small and the overall execution time of such tasks is relatively short, the limitation on the number of processing elements becomes less important.

4.4.2 Task Execution Overhead

Moving tasks and their data sets between machines requires a great deal of overhead on heterogeneous architectures. This overhead is a result of the large amounts of information that must be transmitted over the network plus the cost of the encoding and decoding of data as it passes from one architecture to another.

A study performed in [8] compared PVM and MPI message transfer characteristics over a 10Mbit/s Ethernet network in homogeneous and heterogeneous environments. The homogeneous network was able to use 86% and 74% of the effective network bandwidth using PVM and MPI respectively using 120 kilobyte messages. This resulted in transfer times of 132 ms for PVM and 153 ms for MPI. On the heterogeneous

network, the increased overhead reduced the use of effective bandwidth to 54% for PVM and 43% with corresponding transfer times of 210 ms and 265 ms respectively. Due to the great amount of overhead, the task size must remain large so that the communication-to-computation ratio remains favorable.

In an mHC environment, communication overhead is greatly reduced since all of the devices are directly connected via the PCI bus. A 64-bit PCI bus running at 66 MHz has a theoretical throughput of 4.2 Gb/s [24] which is more then twice the throughput of even the highest performance interconnects being used today which reach as high as 1.92 Gb/s [16]. The overhead created by the encoding and decoding of transmitted data is also greatly reduced since all of the processing elements are designed to be used on the PCI bus and are able to directly access main memory when it is required. This tightly coupled nature allows for even very small tasks to execute efficiently with very low communication-to-computation ratios.

4.4.3 Cost Effectiveness

The cost of deploying a microHeterogeneous computing environment is much less then a standard heterogeneous environment. The processing elements required for mHC cost only hundreds of dollars and can be added to any workstation. This low cost makes a cluster of these machines a very viable and cost effective solution. In comparison, the machines used in a heterogeneous environment can cost tens of thousands of dollars each, and require the extra expense of the high-speed, low latency interconnects to achieve acceptable performance.

4.4.4 Analytical Benchmarking and Profiling

Analytical benchmarking is used for the same purpose in both computing environments. The capabilities of each processing element or machine must be known before program execution begins so the scheduling algorithm is able to determine an efficient mapping of tasks. Since all of the processing elements have different capabilities, scheduling would be impossible without knowing the specific capabilities of each.

While analytical benchmarking is still required, code profiling is not. Since microHeterogeneous computers use a real-time dynamic scheduler that operates during run-time, there is no need to determine the types of processing an application uses during compile time. This eliminates the need for special profiling tools and removes a step from the typical heterogeneous development cycle.

4.4.5 Scheduling Algorithms

Scheduling algorithms in heterogeneous environments generally take place during the compilation stage instead of during execution. In order to create acceptable mappings all of the tasks in an application need to be identified by a profiler and their relative execution times need to be determined. The scheduler is then required to take the list of tasks and map them to the existing hardware.

The microHeterogeneous environment, however, poses new challenges. The task are smaller and contain more dependencies that must be taken into account. Also, the set of tasks an application produces is not known at compile-time. This makes standard heterogeneous scheduling algorithms inadequate. The scheduler for an mHC environment must be dynamic and map tasks in real-time in order to provide the best

performance.

The next chapter discusses the implementation details of the framework that was created in order to support a microHeterogeneous computing environment. The scheduling algorithms that were designed are also examined.

Chapter 5

MICROHETEROGENEOUS COMPUTING FRAMEWORK

The chapter describes the usage and implementation of the microHeterogeneous Computing framework. The different scheduling heuristics that were examined are also discussed.

5.1 Usage

Using the microHeterogeneous computing framework that was implemented is extremely straightforward. A user application needs to include $mhc_api.h$ in order to gain access to the mHC API functions and then link against libmhc. After calling *mhc_initialize* with the proper parameters, the GSL-compatible functions defined in the mHC API can be used to create tasks that will be automatically scheduled on the available mHC compatible devices. The only consideration that needs to be made is that mhc_join should be called before any of the values calculated by an mHC function call is used by a non-mHC instruction. This assures that the function computing this value has completed its execution. An example application is shown in Appendix C.

5.1.1 Initialization Parameters

The *mhc_initialize* API call takes a list of parameters as its argument that are used to initialize the framework. This function must be called before any other API calls

are made. The following is a list of the valid parameters that be used:

-a *value* Sets the value of α to *value*. This value is used in scheduling heuristics to determine the minimum speedup that must be achieved before a task will be scheduled onto a processing element. The default value of α is 1.25 which indicates a speedup of 25% is required.

-b *filename* Use *filename* as the bus configuration file. A bus configuration file must be specified if a scheduler is going to be used (ie a scheduler other then 'none').

-d *filename* Use *filename* as the device configuration file. A device configuration file must be specified if a scheduler is going to be used(ie a scheduler other then 'none').

-g *value* Sets the value of γ to *value*. This value is used in the Weighted Real-Time Min-Min scheduling heuristic. The default value of γ is 0.25.

-l *filename* Write log events to *filename*. If this is not specified, no log information will be recorded.

-r *value* Sets the value of ρ to *value*. This value is used in the Weighted Real-Time Min-Min scheduling heuristic. The default value of ρ is 0.5.

-s *id* Use the scheduling heuristic indicated by *id* to perform the scheduling of tasks. The possible values are:

- **-1** Indicates no scheduler should be used. This results in the application to execute sequentially. No bus configuration or device configuration is required.
- **0** Indicates the Fast Greedy scheduling heuristic should be used. Bus configuration and device configuration files must also be specified.
- **1** Indicates the Real-Time Min-Min scheduling heuristic should be used. Bus configuration and device configuration files must also be specified.
- **2** Indicates the Weighted Min-Min scheduling heuristic should be used. Bus configuration and device configuration files must also be specified.

-t *value* Determines whether the application should execute in normal mode ($value = 0$) or timing mode ($value = 1$). In normal mode, tasks are executed by simulated devices on the host processor and will produce accurate results. Since tasks are actually being executed by the host processor, it is impossible for them to

demonstrate the timing characteristics indicated in the device configuration file. In timing mode, task execution is replaced with sleep statements that represent the expected execution time of the task. In this manner, tasks can appear to execute faster then the host processor would normally execute them.

At an absolute minimum, the scheduler must be set to -1 using '-s -1'. This will have the effect of running the application sequentially, but it will run without errors.

5.1.2 Configuration

The mHC framework is extremely flexible and is able to model any combination of devices on any combination of buses. The configuration is done though the use of XML based configuration files that specify the properties of the devices and buses that are being modelled. Sample configuration files can be found in Appendix B.

Device Configuration

The device configuration file defines each of the devices that is available in the mHC environment. The file contains a list of devices each with their own attributes. These attributes include the device properties, the bus the device uses, and a list of all of the functions that the device supports. The function lists are unique to each device and contains the function ids, the function names, the expected speedup compared to the host processor, and the expected completion given in microseconds per byte of input of each function. The complete BNF for the device configuration file is shown below.

<device config file> ::= **<mHCDeviceConfig>** <device list> **</mHCDeviceConfig>**
<device list> ::= <device> | <device list> <device>
<device> ::= **<Device>** <id> <name> <device desc> <bus> <bus id> <api list> **</Device>**
<device desc> ::= **<Description>** <string> **</Description>**
<bus> ::= **<BusName>** <string> **</BusName>**
<bus id> ::= **<BusID>** <integer> **</BusID>**

<api list> ::= **<APISupport>** <function list> **</APISupport>**
<function list> ::= <function> | <function list> <function>
<function> ::= **<Function>** <id> <name> <speedup> <completion time> **</Function>**
<id> ::= **<ID>** <number> **</ID>**
<name> ::= **<Name>** <string> **</Name>**
<speedup> ::= **<Speedup>** <integer> **</Speedup>**
<completion time> ::= **<CompletionTime>** <double> **</CompletionTime>**
<integer> ::= /* an integer value */
<double> ::= /* a double value */
<string> ::= /* a string value */

The flexibility of the device configuration file allows each device to support different function calls with different speedups. There is no defined limit to the number of devices that can exist or the number of functions that a particular device can support. The nature of XML also allows custom tags to be added to the file if desired, without breaking any functionality when using the file with the mHC framework.

Bus Configuration

The bus configuration file defines each of the buses that are used by the devices. The file contains a list of buses each with their own attributes. These attributes include the bus properties and three important timing values. These timing values include the initialization time of the bus, the per bus transaction overhead, and the per byte transfer time. Each of these is given in microseconds. The complete BNF for the device configuration file is shown below.

<bus config file> ::= **<mHCBusConfig>** <bus list> **</mHCBusConfig>**
<bus list> ::= <bus> | <bus list> <bus>
<bus> ::= **<Bus>** <id> <name> <desc> <init time> <overhead> <transfer> **</Bus>**
<id> ::= **<ID>** <number> **</ID>**
<name> ::= **<Name>** <string> **</Name>**
<desc> ::= **<Description>** <string> **</Description>**
<init time> ::= **<InitTime>** <double> **</InitTime>**
<overhead> ::= **<Overhead>** <double> **</Overhead>**

<transfer> ::= **<TransferTime>** <double> **</TransferTime>**
<integer> ::= /* an integer value */
<double> ::= /* a double value */
<string> ::= /* a string value */

There is no defined limit to the number of buses that can exist, but every bus id that is used in the device configuration file must be defined in the bus configuration file. The nature of XML also allows custom tags to be added to the file if desired, without breaking any functionality when using the file with the mHC framework.

5.2 Implementation

The microHeterogeneous computing framework is a dynamically linked library written purely in C that user applications interact with by way of the mHC API. The framework creates tasks from the API function calls and schedules them to the available processing elements. Currently the library has only been compiled for Linux workstations running the 2.4 kernal.

5.2.1 Overview

After the framework has been successfully initialized, there are three main phases that every task passes through in order to go from an API call to executing on one of the available devices. These three phases are: task creation, task scheduling, and task execution. A high level diagram of the mHC framework is shown in Fig. 5.1.

5.2.2 Initialization

The mHC framework must be initialized before any of the mHC function calls can be used. This is accomplished by using the $mhc_initialize()$ function call. It is during the initialization phase that all of the configuration files are read and the data structures

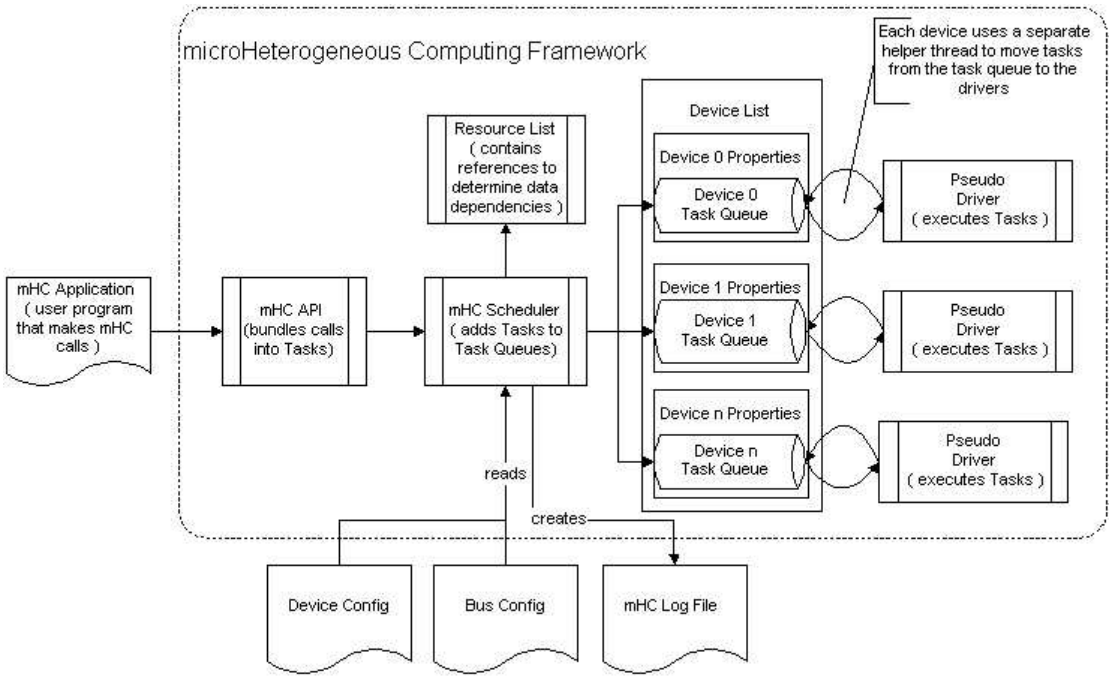

Figure 5.1: microHeterogeneous Computing Framework

required are created. These structures include the device list, the bus list, and the resource list.

A set of helper threads is also created during initialization. These helper threads are used during the task execution phase in order to move tasks from a device's task queue to the device driver so that the task may be executed. If the 'no scheduler' option is chosen, none of this initialization takes place and instead the tasks are simply passed directly to the device driver for execution.

Device List

The device list is a singly-linked list that contains all of the available devices. The list is initialized using the device configuration file that is specified in the *mch_initialize*()

function. This is the only way that devices may be added to the device list. The device list contains pointers to *mDevice* data structures which are defined by:

```
typedef struct mDevice {
    int id;
    char *name;
    char *desc;
    int bus;
    double *speedup;
    int numCalls;
    double *completionTime;
    mQueuePtr taskQueue;
} mDevice, *mDevicePtr;
```

where *id* is the unique id for this device, *name* is the name of the device, and *desc* is a short description. *Bus* is the id of the bus that this device uses. This id must be defined in the bus list. *Speedup* is an array where $speedup_i$ is the speedup that this device delivers over the host processor when executing the function with id i. A value of zero in the *speedup* array indicates that the function is not supported. The total number of function calls that are supported by the device is held in $numCalls$. $CompletionTime$ is an array where $completionTime_i$ is the expected completion time of the function with id i measured in microseconds per operand byte.

Each device also contains its own *taskQueue* which is a priority-based heap in which the tasks scheduled for this device are stored. The heap assures that the task with the highest priority is always the first one to removed. The priority is based on the task ID so that the tasks are removed from the heap in ascending order, regardless of what order they where placed in the heap. This is important when the user application is multi-threaded as tasks may not be scheduled in the same order that they were assigned. Deadlock could occur if a task is dependent on s task that is scheduled after it.

Bus List

The bus list is a singly-linked list that contains the bus definitions that are used by the devices. It is used to determine the time required to transmit a task to a device using a particular bus. The list is initialized using the bus configuration file that is specified in the *mhc_initialize*() function. This is the only way that buses may be added to the bus list. The list contains pointers to *mBus* data structures which are defined by:

```
typedef struct mBus {
    int id;
    char *name;
    char *desc;
    double init;
    double overhead;
    double transfer;
} mBus, *mBusPtr;
```

where *id* is the unique id for the bus, *name* is the name of the bus, *desc* is a short description, *init* is the initialization time of the bus in microseconds, *overhead* is the per transaction overhead in microseconds, and $transfer$ is the transfer time per byte in microseconds.

It is important that every bus id that is used during the device configuration is defined in the bus configuration file. Also, the bus id's must start at 0 and increment by one for each additional bus. This restriction was necessary in order to increase performance within the framework.

Resource Lists

The mHC framework maintains a resource list for the function operands and another list for the function results. They are both singly-linked lists and are used in the task execution phase to check for data dependencies before a task is executed. The resource lists are composed of *mResource* data structures which are defined by:

```
typedef struct mResource {
    int id;
    int deviceID;
    int taskID;
    void *memory;
    size_t size;
} mResource, *mResourcePtr;
```

where *id* is the unique id for the resource, $deviceID$ is the id of the device that needs the resource, $taskID$ is the id of the task associated with the resource, *memory* is a pointer to the beginning of the data block in memory that this resource consists of, and *size* is the size of the data block in memory. The resource list is sorted by $taskID$. This reduces the amount of items that need to be searched when looking for dependencies, since the search can cease as soon as a $taskID$ is found that is greater then or equal to the taskID of the task that is being checked for dependencies.

5.2.3 Task Creation

A task is created every time an mHC API call is made other then $mhc\_initialize()$, $mhc\_finalize()$, and $mhc\_join()$. The tasks encapsulate the relevant information about the function call so that it may be passed to the scheduler in order to get mapped onto an mHC device. This encapsulation is defined by:

```
typedef struct mTask {
    int id;
    int functionID;
    void *args[10];
```

```
    void *ops[10];
    size_t opsSize[10];
    void *res[10];
    size_t resSize[10];
    double etc;
} mTask, *mTaskPtr;
```

The *id* is a unique id for this task and $functionID$ is the id of the mHC API function that was called. The arguments to the function are held in the void pointer array $args$. The arguments that are operands for the function call are held in the void pointer array *ops* with their corresponding sizes in the $opsSize$ array. The arguments that are used to return the results of the function are stored in the void pointer array res with their corresponding sizes in the $resSize$ array. All of these arrays have a static size of ten, where one of the elements must be NULL. This limits the number of arguments that can be passed in an mTask to nine. Finally, the estimated time to completion is held in *etc*. This value is calculated during the scheduling phase.

The $args$ list is used later to obtain pointers to the original arguments of the function call. The *ops* list and the res list are used to determine task dependencies during the task execution phase. These lists actually contain pointers to the block of data specified by the arguments, not the arguments themselves, which may be modified during execution of the function. This is done to handle cases such as matrix views where two different matrix views may actually point to the same block of data.

5.2.4 Task Scheduling

After a task has been created, it is passed into the scheduling phase. When the task enters the scheduling phase, the memory locations specified in the *ops* and res lists are added to the appropriate resource list. The scheduler is then invoked and the

task is placed into the most appropriate task queue. The particulars of each of the different scheduling heuristics are discussed in the following section. After the task has been scheduled, the function call returns allowing the main program to continue execution.

5.2.5 Task Execution

The final phase in the framework is task execution. Task execution is performed by the helper threads that are created during initialization. The threads are lightweight pthreads that use a real-time round-robin scheduling scheme with a slightly higher priority then the main program execution thread. Every device has its own helper thread that moves tasks from the task queue to the device.

If there is a task in the task queue, the helper thread checks to see if the task is dependent on any other tasks that have not yet finished executing. If there is such a dependent task, the thread blocks and allows the other threads in the system to run. Otherwise, the helper thread removes the task from the task queue and passes it to the device driver in order to be executed. Currently, the device driver and execution are simulated and the actual execution of the task is done on the host processor.

If the timing mode option is selected when the framework is initialized, tasks are not actually executed at all. Instead, the helper thread that would have normally executed the task sleeps for the expected completion time of the task. This allows a task to 'execute' faster then is normally possible on the host processor and also allows multiple tasks to be 'executing' simultaneously. The host processor executes a busy loop for the expected completion time of the task. This correctly models how

tasks would actually execute in a mHC environment. Tasks executing on the external processing elements are allowed to execute in parallel and in the correct amount of time independent of the host processor. The host processor, however, blocks execution of the main program when executing tasks by using a busy loop since these events are occurring on the same hardware.

5.3 Scheduling Heuristics

The scheduling heuristic is responsible for taking the tasks created when an API call is made and place it in the task queue of the device that will minimize execution time. The heuristics that are used by the mHC environment are dynamic and real-time which means the mapping is done during run-time and the scheduler only has knowledge about those tasks that have already been scheduled.

The heuristics assume that the target microHeterogeneous environment consists of a set Q of q heterogeneous processing elements. W is a computation cost matrix of size $v \times q$ that contains the estimated execution time for all tasks already created where v is the number of the task currently being scheduled. The value $w_{i,j}$ gives the estimated execution time of task i on processing element p_i. B is a communication cost matrix of size $q \times 2$, where $b_{i,1}$ is the communication time required to transfer this task to processing element p_i and $b_{i,2}$ is the per transaction overhead for communicating with processing element p_i. The estimated time to completion (ETC) of task i on processing element p_j is defined as

$$ETC_{i,j} = w_{i,j} + b_{j,1} + b_{j,2} \tag{5.1}$$

The estimated time to idle is the estimated amount of time before a processing element

p_i will become idle. The estimated time to idle (ETI) of processor j is defined as

$$ETI_j = \sum_{1 \leq i \leq n} ETC_{i,j} \tag{5.2}$$

where n is the number of tasks currently scheduled on processing element i. This does not take into account any processing that might have already been completed on the first task in the queue. However, since the number of tasks created is reasonably large and the execution times of the tasks is relatively short this does not have a negative impact on the performance of the scheduling heuristics.

Three different schedulers were implemented in order to compare their performances. All of these algorithms are based on heterogeneous scheduling algorithms which have been modified to fit the requirements of a microHeterogeneous environment if required.

5.3.1 Fast Greedy

The Fast Greedy algorithm is a very simple algorithm that simply searches for the processor which has the lowest ETC for the task that is being scheduled. Tasks that have previously been scheduled are not taken into account at all in this algorithm.

The Fast Greedy algorithm first determines the subset of processors, S, of Q that support the current task being scheduled. The ETC for the task on each of the processors in S is then calculated. The processor, s_j, with the minimum ETC is chosen and compared against the ETC of the host processor. If the speedup gained is greater then γ then the task is scheduled to s_j, otherwise the task is scheduled to the host processor. The complete algorithm is shown below.

Fast Greedy

1. Let i represent the current task to be scheduled.
2. Determine that set of processors S of size s that support the function call c_i where c_i is the function that is to be executed. The host processor is denoted as s_0 and must always appear in S.
3. Calculate the ETC of task i for each of the s processors.
4. Locate the processor s_j where
$$ETC_{i,s_j} = \min_{0 \leq j \leq s} ETC_{i,j}$$
5. If $ETC_{i,s_j}/ETC_{i,s_0} > \gamma$ then schedule task i on processor s_j otherwise schedule task i on processor s_0

The parameter γ is used as a final check on whether or not the performance gained is worth the effort to send a task to a processor. This allows some fine turning to prevent very small tasks from being sent to processing elements where the higher communication-to-computation ratio would be undesirable.

5.3.2 Real-Time Min-Min

The Real-Time Min-Min (RTmm) algorithm is based on the standard Min-Min algorithm that is used in heterogeneous computing. The Min-Min algorithm was chosen because it is simple, very fast, and generally results in good performance. The Min-Min algorithm had to be modified because information about all of the tasks is not known at run-time, only information about previously created tasks is known. Thus tasks are scheduled in order to minimize the expected time to completion as tasks are scheduled.

The RTmm algorithm first determines the subset of processing elemments, S, of Q that support the current task being scheduled. The ETC for the task and the ETI

for each of the processing elements in S is then calculated. The $ETC_{i,j(total)}$ for task i on processing element p_j is equal to the sum of ETC_{i_j} and ETI_j. The processing element, s_j, with the minimum newly calculated ETC_{total} is chosen and compared against the ETC_{total} of the host processor. If the speedup gained is greater then γ then the task is scheduled to s_j, otherwise the task is scheduled to the host processor. The complete algorithm is shown below.

Real-Time Min Min

1. Let i represent the current task to be scheduled.
2. Determine that set of processors S of size s that support the function call c_i where c_i is the function that is to be executed. The host processor is denoted as s_0 and must always appear in S.
3. Compute the ETC of task i for each of the s processors.
4. Compute the ETI of processing element p_j for each of the s processors.
5. Compute $ETC_{i,j(total)} = ETC_{i,j} + ETI$ for each of the s processors.
6. Locate the processor s_j where
$$ETC_{i,s_j(total)} = \min_{0 \leq j \leq s} ETC_{i,j(total)}$$
7. If $ETC_{i,s_j(total)}/ETC_{i,s_0(total)} > \gamma$ then schedule task i on processor s_j otherwise schedule task i on processor s_0

The parameter γ is again used as a final check on whether or not the performance gained is worth the effort to send a task to a processor.

5.3.3 Weighted Real-Time Min-Min

The Weighted Real-Time Min-Min (WRTmm) uses the same algorithm as RTmm but adds two more parameters so that the scheduling can be fine tuned to specific applications. First, the parameter α takes into account times when a task dependency exists for the task that is currently being scheduled. An α value of less then one tends

to schedule these tasks onto the same processing element as the dependency while a value greater then one tends to schedule these tasks onto a different processing element.

Second, the parameter ρ is used to try and schedule events to processing elements that support fewer of the API calls and must be between 0 and 1. A low value of ρ tells the scheduler not worry about mapping tasks to devices that do not support many functions, while the opposite is true for high values of ρ. This parameter is most useful for times when a custom hardware device is being used that only supports a couple of the function calls. Ordinarily, such a device would remain idle a good portion of the time if other devices also supported the same functions. The complete algorithm is shown below.

Weighted Real-Time Min Min

1. Let i represent the current task to be scheduled.
2. Determine that set of processors S of size s that support the mHC function call c_i where c_i is the mHC function that is to be executed. The host processor is denoted as s_0 and must always appear in S.
3. Compute the ETC of task i for each of the s processors.
4. Compute the ETI of processing element p_j for each of the s processors.
5. Compute $ETC_{i,j(total)} = ETC_{i,j} + ETI$ for each of the s processors.
6. Compute $ETC_{weighted}$ for each of the s processes which is defined as

$$ETC_{i,j(weighted)} = (ETC_{i,j(total)} \times \alpha) \times [1 - (\rho \times \frac{1}{c})]$$

   where c is the total number of mHC function calls that p_j supports.
7. Locate the processor s_j where

$$ETC_{i,s_j(weighted)} = \min_{0 \leq j \leq s} ETC_{i,j(weighted)}$$

8. If $ETC_{i,s_j(weighted)}/ETC_{i,s_0(weighted)} > \gamma$ then schedule task i on processor s_j otherwise schedule task i on processor s_0

The parameter γ is again used as a final check on whether or not the performance gained is worth the effort to send a task to a processor.

The next chapter discusses the results that were obtained using each of these scheduling heuristics on various permutations of a microHeterogeneous computing environment.

Chapter 6

RESULTS

This chapter discusses the experimental results of this thesis. All of the results were obtained from running the microHeterogeneous computing framework on a 400 MHz, Pentium II processer workstation running Red Hat Linux 7.3.

6.1 Methodology

In order to assure accurate timing information, each simulation run was done in three steps. The default values for α, γ, and ρ were used for each of the simulation runs. First, the application was executed without using a scheduler causing the application to execute sequentially. The was done five times and the median run was used to determine the estimated completion time of each of the different function types that were used by the application. The estimated completion time of each of these functions was then calculated for each of the available devices based on the speedup provided by the device. Next, the application was run with the desired scheduling heuristic which caused the tasks to actually be mapped to various processing elements. This was done as a check to make sure that the output of the parallelized application was the same as when the application was run sequentially. Finally, the application was run using the desired scheduling heuristic in the timing mode that is provided by the framework. This step does not actually execute the tasks and therefore the resulting output will be different then the sequential version. It does, however, utilize

the timing information calculated during the first step in order to accurately model the timing of a microHeterogeneous environment. This final step was also done five times and the median of the runs was used.

While any combination of devices is possible using the mHC framework, the simulations were made more manageable by only using three different base combinations of devices. These combinations were then altered by changing the number of devices, the speedups associated with the devices, and the bus timing information associated with the device. When the speedups or bus timing was changed, it was changed equally on all devices. The base combinations used were:

default The default combination modelled three PCI-based processing elements which supported all of the mHC API calls with a speedup of 20.

full_same The full_same combination utilized six PCI-based processing elements which supported all of the mHC API calls with a speedup of 20.

full_diff The full_diff combination utilized six PCI-based processing elements which supported all of the mHC API calls with varying speedups. The speedups used were 20,10,5,2,1, and 0.5 for devices one through six respectively. When the number of devices is altered, devices are always removed starting at device six.

The speedup of 20 was decided as a conservative value based on the processing elements that were previously mentioned. These accelerators were actually reported to have speedups in this range as compared to a 1.4 GHz Pentium IV and should actually surpass this mark when compared to 400 MHz processor used for the simulations.

6.2 mHC Simulator

An mHC Simulator was implemented in Qt which provides a graphical user interface for setting up and running simulations. Using the simulator, all of the configuration files that the framework requires can be generated without the need to edit the text files themselves. The simulator also allows simulations to be run that involve various mHC applications, various device configurations, and various bus timing values.

The simulator follows the same procedure outlined in the Methodology section for obtaining timing results without the need for any user intervention. It automatically calculates and records all of the estimated completion times that are required for the simulations, checks the parallel output against the sequential version, and then performs the timing runs.

The simulator also automatically analyzes all of the log files that are generated by the framework and produces in-depth HTML reports. These reports include execution time comparisons, device utilization charts, program statistics, and scheduler performance analysis. Interactive task graphs are also generated that indicate the device to which each task was assigned, what order the tasks were executed in, and also where the task dependencies occurred that resulted in a device being blocked.

6.3 Application Simulations

Three different applications were simulated using the full_same device configuration. The simulations show the performance when simulated on a microHeterogeneous computer for some common operations that are found in scientific computing applications.

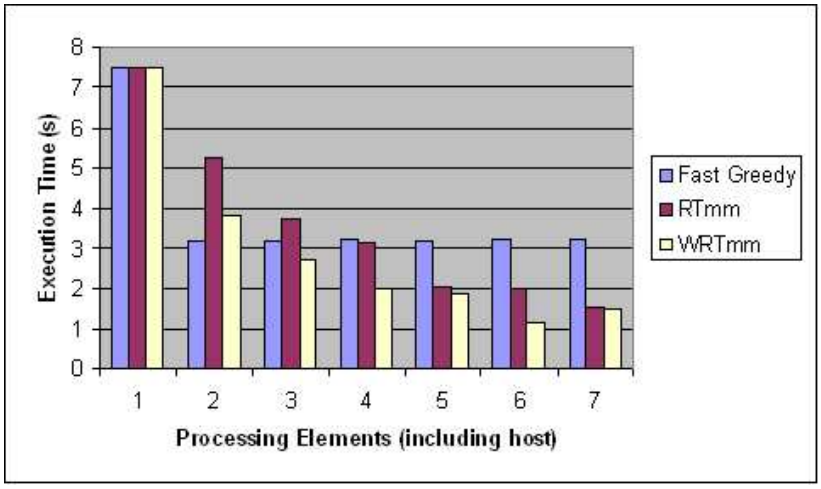

Figure 6.1: Matrix Application: Performance vs Number of Devices

6.3.1 Matrix

The matrix application performs some basic matrix operations on a set of fifty 100 $\times$ 100 matrices. Initially, the first twenty-five matrices are summed together, the last twenty-five matrices are subtracted from one another, and every fifth matrix is scaled by a constant factor. This provides for a good mix of operations and dependencies and represents some of the common operations performed on matrices.

After the basic matrix math operations, the inverse of each of the fifty matrices is determined. This is accomplished by first performing an LU decomposition on the matrix, followed by the matrix inversion. Matrix inversion is a very common task that is relatively computation intensive.

The results of the simulation are shown in Fig. 6.1. The results along with the speedups as compared to the single processor machine are also shown in Table 6.1. All of the different configurations produced a performance improvement over

Heuristic		Number of Processing Elements						
		1	2	3	4	5	6	7
Fast Greedy	Exec Time (s)	7.5	3.22	3.22	3.25	3.2	3.25	3.25
	Speedup	-	2.33	2.33	2.31	2.34	2.31	2.31
RTmm	Exec Time (s)	7.5	5.25	3.75	3.14	2.04	2	1.5
	Speedup	-	1.43	2.00	2.39	3.68	3.75	5.00
WRTmm	Exec Time (s)	7.5	3.81	2.7	2.02	1.89	1.14	1.48
	Speedup	-	1.97	2.78	3.71	3.97	6.58	5.07

Table 6.1: Matrix Simulation Data Summary

the sequential version (indicated in the graph as having only one processing element) to varying degrees. The Fast Greedy heuristic produced a decent speedup, but did not take advantage of the additional devices as they were added. It did, however, produce the best results for the cases of two and three processing elements due to the low scheduling overhead of the heuristic. The RTmm heuristic and the WRTmm both scaled well versus the number of processing elements, with WRTmm consistently providing lower overall execution times. The highest speedup obtained was 6.58 and occurred using WRTmm with six processing elements.

6.3.2 Stats

The stats application divides a block of five million values into fifty blocks of one hundred thousand values and then determines the blocks of data with the minimum and maximum standard deviations. This is accomplished by finding the standard deviation of each of the fifty blocks, storing the values into an array, and then locating the indices of the minimum and maximum values. This application represents the parallel search of a data space for a specific value, in this case the minimum and

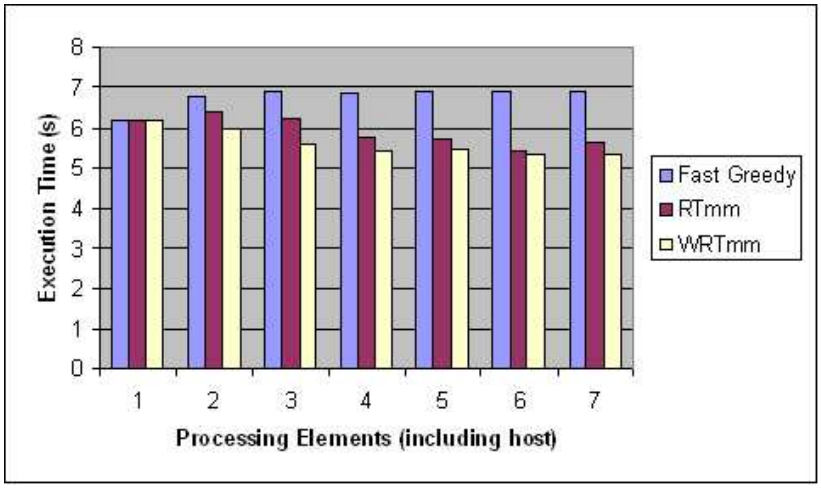

Figure 6.2: Stats Application: Performance vs Number of Devices

maximum standard deviation.

The results of the simulation are shown in Fig. 6.2. The results along with the speedups as compared to the single processor machine are also shown in Table 6.2. For this application, only the WRTmm heuristic consistently produced schedules that produced a performance improvement over the sequential version. The Fast Greedy heuristic produced did not produce an acceptable mapping due to its inability to balance loads effectively. The RTmm heuristic and the WRTmm both produced acceptable mappings, with WRTmm consistently providing lower overall execution times. The highest speedup obtained was 1.16 and occurred using WRTmm with six processing elements. The performance increase was not as dramatic as with the matrix application because the standard deviation function used is not as computationally intensive. This left most of the devices idle for a large portion of the total execution time, greatly reducing their effectiveness.

Heuristic		Number of Processing Elements						
		1	2	3	4	5	6	7
Fast Greedy	Exec Time (s)	6.2	6.78	6.91	6.85	6.9	6.88	6.9
	Speedup	-	0.91	0.90	0.91	0.90	0.90	0.90
RTmm	Exec Time (s)	6.2	6.4	6.23	5.8	5.76	5.4	5.62
	Speedup	-	0.97	1.00	1.07	1.08	1.15	1.10
WRTmm	Exec Time (s)	6.2	6	5.59	5.42	5.44	5.35	5.32
	Speedup	-	1.03	1.11	1.14	1.14	1.16	1.17

Table 6.2: Stats Simulation Data Summary

6.3.3 Linalg

The linalg application solves fifty sets of linear equations each containing one hundred and seventy-five variables. This is another rather computation intensive application that is very common in scientific applications. In order to solve each matrix must first be decomposed and then passed to the solve function, creating a direct dependency for each of the fifty sets of equations.

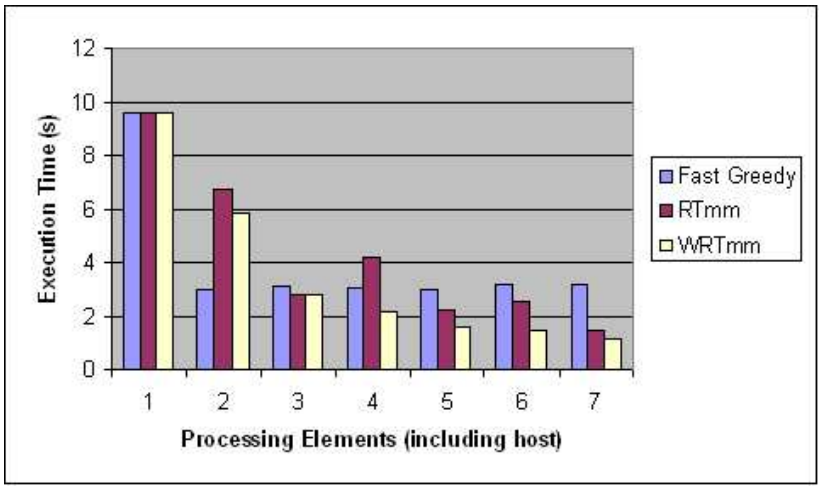

Figure 6.3: Linalg Application: Performance vs Number of Devices

Heuristic		Number of Processing Elements						
		1	2	3	4	5	6	7
Fast Greedy	Exec Time (s)	9.6	3	3.12	3.08	3.02	3.18	3.22
	Speedup	-	3.20	3.08	3.12	3.18	3.02	2.98
RTmm	Exec Time (s)	9.6	6.75	2.82	4.2	2.21	2.58	1.5
	Speedup	-	1.42	3.40	2.29	4.34	3.72	6.40
WRTmm	Exec Time (s)	9.6	5.86	2.82	2.14	1.6	1.49	1.16
	Speedup	-	1.64	3.40	4.49	6.00	6.44	8.28

Table 6.3: Linalg Simulation Data Summary

The results of the simulation are shown in Fig. 6.3. The results along with the speedups as compared to the single processor machine are also shown in Table 6.3. All of the different configurations produced a performance improvement over the sequential version (indicated in the graph as having only one processing element) to varying degrees. The Fast Greedy heuristic produced a decent speedup, but again did not take advantage of the additional devices as they were added. It did, however, produce the best results for the case of two processing elements due to the low scheduling overhead of the heuristic. The performance of the RTmm heuristic was heavily dependent on whether or not the number of processing elements was odd. If the number was odd, the number of times devices had to be blocked due to dependencies was reduced and the heuristic performed better. The WRTmm algorithm consistently providing lower overall execution times and scaled well with the number of processing elements. The WRTmm algorithm was also not dependent on the number of processing elements being odd. The highest speedup obtained was 8.28 and occurred using WRTmm with seven processing elements.

6.4 Scheduler Performance Comparison

A final stress testing was performed on each of the scheduling heuristics to provide a more complete comparison between them. The application used to perform the testing generated one hundred different task graphs with which to test the schedulers on. The task graphs created each contained three hundred tasks with varying depths and a unique set of dependencies between them. The task performed was a simple element multiplication of two matrices. The average execution time of the one hundred task graphs was used to compare the performance of the different schedulers.

The task graphs that were created posed a challenge to the schedulers since the dependencies did not follow any regular pattern. There was no way for the heuristics to obtain a good schedule by chance, which had occurred in one of the applications simulated previously. Also, the task chosen was intentionally not very computationally intensive. The performance achieved was therefore very dependent on the ability of the scheduler to handle the dependencies as well as efficiently balance the load between the processors.

The schedulers were tested with both similar and dissimilar devices. They were also tested with increasing bus transfer times, as well as decreasing device performance in order to determine their effect on overall application performance.

First, the schedulers were tested with the full_same device configuration using various numbers of devices. This simulation was used to test the ability of the heuristics to effectively utilize additional devices by balancing the load equally. The results of the simulation are shown in Fig. 6.4. The results are also shown in Table 6.4.

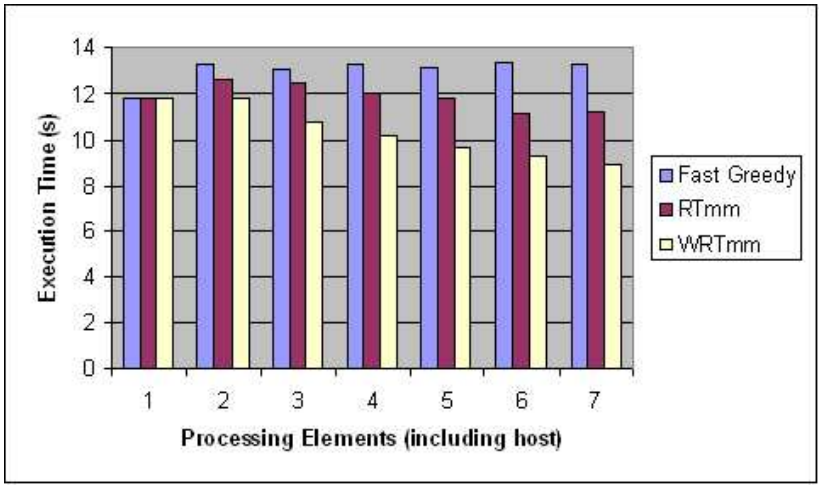

Figure 6.4: Performance of Random Task Graphs on a Uniform Set of Devices

In this test, both the RTmm and WRTmm algorithms reduced their overall execution times as more devices were added. However, the WRTmm algorithm showed a sharper decline in execution time as devices were added as well as better overall performance. The Fast Greedy algorithm performs no load balancing at all and demonstrated a relatively constant execution time that was greater then both the RTmm and WRTmm heuristics.

Second, the schedulers were tested with the full_diff device configuration using

		Number of Processing Elements						
Heuristic		1	2	3	4	5	6	7
Fast Greedy	Exec Time (s)	11.8	13.25	13.1	13.3	13.16	13.32	13.25
RTmm	Exec Time (s)	11.8	12.67	12.51	11.96	11.75	11.16	11.19
WRTmm	Exec Time (s)	11.8	11.75	10.74	10.23	9.6	9.27	8.92

Table 6.4: Uniform Set of Devices Performance Data

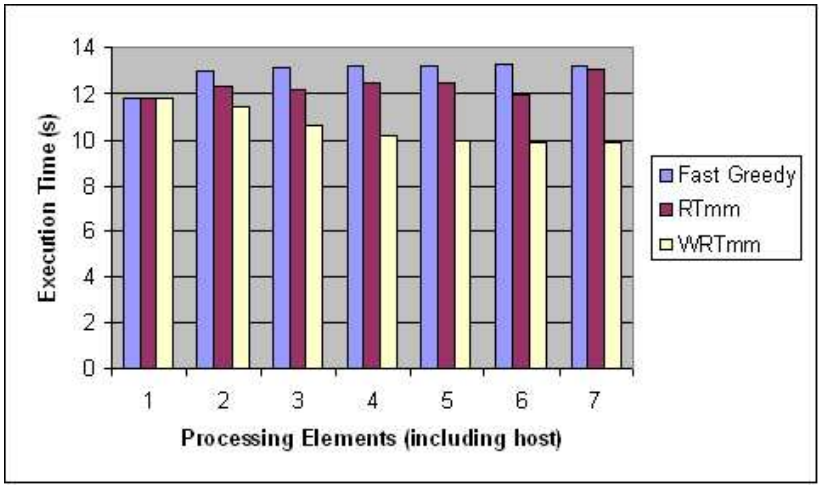

Figure 6.5: Performance of Random Task Graphs on a Non-Uniform Set of Devices

various numbers of devices. This simulation was used to test the ability of the heuristics to effectively utilize additional devices by balancing the load to the most effective processing element. The results of the simulation are shown in Fig. 6.5. The results are also shown in Table 6.5.

In this test, only the WRTmm algorithm reduced the overall execution time as more devices were added. This shows that the WRTmm is able to take advantage of slower devices without having a negative impact on the overall performance. The RTmm, however, was unable to determine a adequate mapping to take advantage

		Number of Processing Elements						
Heuristic		1	2	3	4	5	6	7
Fast Greedy	Exec Time (s)	11.8	13.01	13.12	13.22	13.22	13.29	13.18
RTmm	Exec Time (s)	11.8	12.3	12.11	12.46	12.48	11.92	13.08
WRTmm	Exec Time (s)	11.8	11.4	10.62	10.19	9.98	9.82	9.84

Table 6.5: Non-Uniform Set of Devices Performance Data

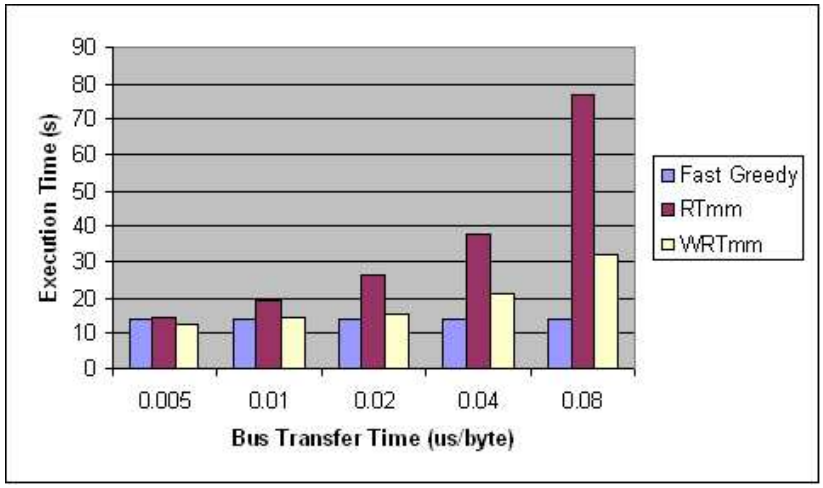

Figure 6.6: Performance of Random Task Graphs for Increasing Bus Transfer Times

of such devices and actually showed a decrease in performance as more devices were added. The Fast Greedy algorithm performs no load balancing at all and demonstrated a relatively constant execution time that was greater then both the RTmm and WRTmm heuristics.

Next, the schedulers were tested with the default device configuration using increasing values for the bus transfer times. This simulation was used to test the ability of the heuristics to cope with higher communication times due to slower buses. The bus transfer times tested start at the level of a 33 MHz 64-bit PCI bus and end at the level of a 100 Mb/s ethernet connection. The results of the simulation are shown in Fig. 6.6. The results are also shown in Table 6.6.

For this test, the Fast Greedy heuristic performed the best and showed not performance degradation as the bus transfer times were increased. This is because the Fast Greedy heuristic simply places the tasks on the device with the lowest estimated

Heuristic		Number of Processing Elements				
		0.005	0.01	0.02	0.04	0.08
Fast Greedy	Exec Time (s)	13.7	13.82	13.71	13.66	13.94
RTmm	Exec Time (s)	14.21	19.53	26.1	37.68	76.54
WRTmm	Exec Time (s)	12.43	14.32	15.01	21.03	31.76

Table 6.6: Increasing Bus Transfer Times Performance Data

time to completion which is always the host processor as the bus transfer times are increased. The RTmm and WRTmm algorithms both showed increased execution times as the bus transfer times increased. Overall, the WRTmm algorithm performed better then the RTmm algorithm and the execution time only increased by a factor of three while the bus transfer times increased by a factor of forty. In comparison, the execution time using the RTmm algorithm increased by a factor of approximately six.

Finally, the schedulers were tested with the default device configuration but with varying device speedups. This simulation was used to test the ability of the heuristics to perform well even with slower devices. The device speedups tested start at the 4 and end at 40. The results of the simulation are shown in Fig. 6.7. The results are also shown in Table 6.7.

Heuristic		Number of Processing Elements				
		4	5	10	20	40
Fast Greedy	Exec Time (s)	17	16.27	13.75	14.18	13.12
RTmm	Exec Time (s)	14.3	13.8	11.9	11.8	11.6
WRTmm	Exec Time (s)	11.26	11.23	10.76	10.75	10.61

Table 6.7: Varying Speedup Performance Data

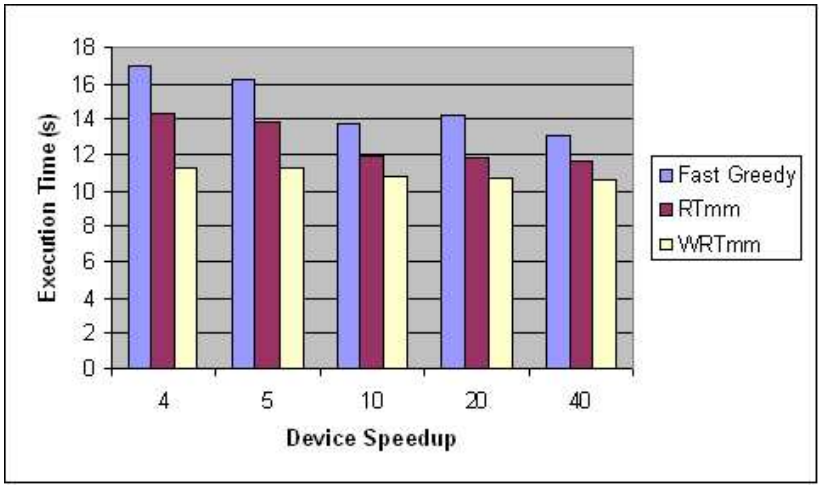

Figure 6.7: Performance of Random Task Graphs for Varying Device Speedups

In this test, both the Fast Greedy and RTmm algorithms proved to be very dependent on the speedup of the devices in order to achieve good performance. Each time the performance of the devices was reduced by half, the execution time for both the Fast Greedy algorithm and the RTmm algorithm would increase by approximately a second. The WRTmm algorithm, however, did not demonstrate this dependence at all and the execution time changed very little as the device speedup was reduced. This is close to the ideal case since the task graphs were not composed of computationally intensive tasks whose execution time could be improved by any dramatic amount by increasing the device speedup alone. It is instead the inefficient load balancing of the Fast Greedy and RTmm algorithms that actually gets partially hidden by faster devices in this particular case.

Overall, the WRTmm algorithm proved to have the best scheduling performance in almost every case. The more basic RTmm was never able to perform as well as the WRTmm algorithm in any of the test that were performed. The Fast Greedy algo-

rithm also generally showed worse performance then the WRTmm algorithm, except in a few special cases. For instance, when only two processing elements (including the host processor) is going to be used, the Fast Greedy algorithm provides good performance due to its very low scheduling time.

These results show that the microHeterogeneous computing environment is a viable architecture that can provide measurable performance benefits in almost all cases. The next chapter discusses the accomplishments, limitations, and future work in regards to this thesis.

Chapter 7

CONCLUSIONS

7.1 Accomplishments

This thesis presents a new computing paradigm, microHeterogeneous Computing, that successfully exploits fine-grained parallelism in scientific based applications using additional PCI based processing elements. An API was created that allows developers to incorporate mHC into their applications without being required to address task scheduling, load balancing, or threading issues. A highly configurable mHC framework was implemented as a standard library which allows actual mHC compliant applications to be compiled and executed using standard techniques. All of the mHC devices were simulated as no actual mHC compliant devices exist at this time. The performance results obtained showed the mHC environment to be a viable computing architecture that has the capability to measurably increase performance over a comparable single processor machine.

An mHC simulator with a graphical user interface was also implemented. The mHC simulator has the ability to generate all of the configuration files that are required by the mHC framework. It also allows the user to automatically run simulations for mHC applications with various user-defined configurations. Full featured HTML based reports are then created based on the log files generated from the sim-

ulation.

7.2 Limitations

The main limitation of this work is the lack of real mHC compliant drivers. As no devices of this nature currently exist, they had to be simulated which creates the possibility of discrepancies between the simulated performance and the actual performance of a true mHC environment. Every effort was made to accurately model device interactions, however, not every aspect that effects overall performance could be properly accounted for.

7.3 Future Work

There is still much work to do in the field of microHeterogeneous computing. All aspects of the architecture need to be refined in order to achieve the highest possible performance. The area that requires the most intensive effort is the creation of mHC compliant device drivers so that an actual mHC environment can be created. Suitable devices already exist, however there are currently no drivers to connect them to the mHC library so that function calls may be mapped to them. Once device drivers are created, the framework can simply be modified to utilize these drivers instead of executing the function calls internally.

Dynamic real-time scheduling algorithms must also be studied in order to obtain better mappings in this type of environment. This architecture presents some new challenges by combining the issues of mapping tasks onto heterogeneous processing elements with the issues of real-time scheduling in a multiprocessor system. While a cursory algorithm was presented here, more work needs to be done especially when

an actual mHC test environment becomes available.

While the microHeterogneous API currently contains the most common scientific functions, it needs to be expanded in order to become complimentary to the GNU Scientific Library. This will further ease the transition for developers moving from GSL applications to mHC applications.

Finally, the concept of mHC clusters needs to be fully explored in order to determine the applicability of mHC to this area of computing. By exploiting both the coarse-grained and fine-grained parallelism, performance should dramatically increase, but this hypothesis needs to tested in a real-world situation.

Appendix A

COMPLETE MHC API

The following is a complete list of currently supported mHC API calls. All of the API calls return 0 if successful and a negative error code otherwise. The mHC library is thread-safe except for $mHC\_initialize(...)$ and $mHC\_finalize()$. These two functions should only be called by the main application thread at the very beginning and very end of execution respectively.

A.1 mHC Specific Functions

mhc_initialize(char **argv)
This function initializes the microHeterogeneous API and must be called before any of the API functions may be called. The arg list passed to the application should be passed into mhc_initialize as well in order to be fully compliant with the mHC Simulator.

mhc_join()
This function waits for all currently scheduled tasks to finish before returning. This function should be called before using any result from an mHC API call is used by a non mHC API call.

mhc_finalize()
This function cleans up the microHeterogeneous API and mustbe called before exiting to assure that all memory has been properly freed and all file handles have been properly closed.

A.2 Matrix Operations

mhc_matrix_add(gsl_matrix *a, gsl_matrix *b)
This function adds the elements of matrix b to the elements of matrix a, $a'(i,j) = a(i,j) + b(i,j)$. The two matrices must have the same dimensions.

mhc_matrix_sub(gsl_matrix *a, gsl_matrix *b)
This function subtracts the elements of matrix b from the elements of matrix a, $a'(i,j) = a(i,j) - b(i,j)$. The two matrices must have the same dimensions.

mhc_matrix_mul_elements(gsl_matrix *a, gsl_matrix *b)
This function multiplies the elements of matrix a by the elements of matrix b, $a'(i,j) = a(i,j)b(i,j)$. The two matrices must have the same dimensions.

mhc_matrix_div_elements(gsl_matrix *a, gsl_matrix *b)
This function divides the elements of matrix a by the elements of matrix b, $a'(i,j) = a(i,j)/b(i,j)$. The two matrices must have the same dimensions.

mhc_matrix_scale(gsl_matrix *a, double *x)
This function multiplies the elements of matrix a by the constant factor x, $a'(i,j) = xa(i,j)$.

mhc_matrix_add_constant(gsl_matrix *a, double *x)
This function adds the constant value x to the elements of the matrix a, $a'(i,j) = a(i,j) + x$.

mhc_matrix_minmax(gsl_matrix *m, double *min_out, double *max_out)
This function returns the minimum and maximum values in the matrix m, storing them in min_out and max_out.

mhc_matrix_minmax_index(gsl_matrix *m, size_t *imin, size_t *jmin, size_t *imax, size_t *jmax)
This function returns the indices of the minimum and maximum values in the matrix m, storing them in (imin,jmin) and (imax,jmax). When there are several equal minimum or maximum elements then the first elements found are returned.

A.3 Vector Operations

mhc_vector_add(gsl_vector *a, gsl_vector *b)
This function adds the elements of vector b to the elements of vector a, $a'\_i = a\_i + b\_i$. The two vectors must have the same length.

mhc_vector_sub(gsl_vector *a, gsl_vector *b)
This function subtracts the elements of vector b from the elements of vector a, $a'\_i = a\_i - b\_i$. The two vectors must have the same length.

mhc_vector_mul(gsl_vector *a, gsl_vector *b)
This function multiplies the elements of vector a by the elements of vector b, $a'\_i = a\_ib\_i$. The two vectors must have the same length.

mhc_vector_div(gsl_vector *a, gsl_vector *b)
This function divides the elements of vector a by the elements of vector b, $a'\_i = a\_i/b\_i$. The two vectors must have the same length.

mhc_vector_scale(gsl_vector *a, double *x)
This function multiplies the elements of vector a by the constant factor x, $a'\_i = xa\_i$.

mhc_vector_add_constant(gsl_vector *a, double *x)
This function adds the constant value x to the elements of the vector a, $a'\_i = a\_i + x$.

mhc_vector_minmax(gsl_vector *m, double *min_out, double *max_out)
This function returns the minimum and maximum values in the vector v, storing them in min_out and max_out.

mhc_vector_minmax_index(gsl_vector *m, size_t *imin, size_t *imax)
This function returns the indices of the minimum and maximum values in the vector v, storing them in imin and imax. When there are several equal minimum or maximum elements then the lowest indices are returned.

A.4 Polynomial Solve

mhc_poly_complex_solve(double *a, size_t *n, gsl_poly_complex_workspace *w, gsl_complex_packed_ptr z)
This function computes the roots of the general polynomial $P(x) = a\_0 + a\_1x + a\_2x^2 + ... + a\_n - 1x^{n-1}$ using balanced-QR reduction of the companion matrix. The parameter n specifies the length of the coefficient array. The coefficient of the highest order term must be non-zero. The function requires a workspace w of the appropriate size. The $n - 1$ roots are returned in the packed complex array z of length $2(n - 1)$, alternating real and imaginary parts.

A.5 Permutations

mhc_permutation_reverse(gsl_permutation *p)
This function reverses the elements of the permutation p.

mhc_permutation_inverse(gsl_permutation *inv, gsl_permutation *p)
This function computes the inverse of the permutation p, storing the result in inv.

mhc_permutation_next(gsl_permutation *p)
This function advances the permutation p to the next permutation in lexicographic order and returns GSL_SUCCESS. If no further permutations are available it leaves p unmodified. Starting with the identity permutation and repeatedly applying this function will iterate through all possible permutations of a given order.

mhc_permutation_prev(gsl_permutation *p)
This function steps backwards from the permutation p to the previous permutation in lexicographic order. If no previous permutation is available it leaves p unmodified.

mhc_permute(size_t *p, double *data, size_t *stride, size_t *n)
This function applies the permutation p to the array data of size n with stride stride.

mhc_permute_inverse(size_t *p, double *data, size_t *stride, size_t *n)
This function applies the inverse of the permutation p to the array data of size n with stride stride.

mhc_permute_vector(gsl_permutation *p, gsl_vector *v)
This function applies the permutation p to the elements of the vector v, considered as a row-vector acted on by a permutation matrix from the right, $v' = vP$. The j-th column of the permutation matrix P is given by the p_j-th column of the identity matrix. The permutation p and the vector v must have the same length.

mhc_permute_vector_inverse(gsl_permutation *p, gsl_vector *v)
This function applies the inverse of the permutation p to the elements of the vector v, considered as a row-vector acted on by an inverse permutation matrix from the right, $v' = vP^T$. Note that for permutation matrices the inverse is the same as the transpose. The j-th column of the permutation matrix P is given by the p_j-th column of the identity matrix. The permutation p and the vector v must have the same length.

mhc_permutation_mul(gsl_permutation *p, gsl_permutation *pa, gsl_permutation *pb)
This function combines the two permutations pa and pb into a single permutation p, where p = pa . pb. The permutation p is equivalent to applying pb first and then pa.

A.6 Combinations

mhc_combination_next(gsl_combination *c)
This function advances the combination c to the next combination in lexicographic order. If no further combinations are available it leaves c unmodified. Starting with the fisrst combination and repeatedly applying this function will iterate through all possible combinations of a given order.

mhc_combination_prev(gsl_combination *c)
This function steps backwards from the combination c to the previous combination in lexicographic order. If no previous combination is available it leaves c unmodified.

A.7 Sorting

mhc_heapsort(void *array, size_t *count, size_t *size, gsl_comparison_fn_t compare)
This function sorts the count elements of the array array, each of size size, into ascending order using the comparison function compare. The type of the comparison function is defined by, int (*gsl_comparison_fn_t) (voida, voidb) A comparison function should return a negative integer if the first argument is less than the second argument, 0 if the two arguments are equal and a positive integer if the first argument is greater than the second argument.

A.8 Linear Algebra

mhc_linalg_LU_decomp(gsl_matrix *A, gsl_permutation *p, int *signum)
These functions factorize the square matrix A into the LU decomposition $PA = LU$. On output the diagonal and upper triangular part of the input matrix A contain the matrix U. The lower triangular part of the input matrix (excluding the diagonal) contains L. The diagonal elements of L are unity, and are not stored. The permutation matrix P is encoded in the permutation p. The j-th column of the matrix P is given by the k-th column of the identity matrix, where $k = p\_j$ the j-th element of the permutation vector. The sign of the permutation is given by signum. It has the value $(-1)^n$, where n is the number of interchanges in the permutation. The algorithm used in the decomposition is Gaussian Elimination with partial pivoting (Golub & Van Loan, Matrix Computations, Algorithm 3.4.1).

mhc_linalg_complex_LU_decomp(gsl_matrix_complex *A, gsl_permutation *p, int *signum)
Complex version of mhc_linalg_LU_decomp.

mhc_linalg_LU_solve(gsl_matrix *LU, gsl_permutation *p, gsl_vector *b, gsl_vector *x)

These functions solve the system $Ax = b$ using the LU decomposition of A into (LU, p) given by gsl_linalg_LU_decomp or gsl_linalg_complex_LU_decomp.

mhc_linalg_complex_LU_solve(gsl_matrix_complex *LU, gsl_permutation *p, gsl_vector_complex *b, gsl_vector_complex *x)
Complex version of mhc_linalg_LU_solve.

mhc_linalg_LU_svx(gsl_matrix *LU, gsl_permutation *p, gsl_vector *x)
These functions solve the system $Ax = b$ in-place using the LU decomposition of A into (LU,p). On input x should contain the right-hand side b, which is replaced by the solution on output.

mhc_linalg_complex_LU_svx(gsl_matrix_complex *LU, gsl_permutation *p, gsl_vector_complex *x)
Complex version of mhc_linalg_LU_svx.

mhc_linalg_LU_refine(gsl_matrix *A, gsl_matrix *LU, gsl_permutation *p, gsl_vector *b, gsl_vector *x, gsl_vector *residual)
These functions apply aniterative improvement to x, the solution of $Ax = b$, using the LU decomposition of A into (LU, p). The initial residual $r = Ax - b$ is also computed and stored in residual.

mhc_linalg_complex_LU_refine(gsl_matrix_complex *A, gsl_matrix_complex *LU, gsl_permutation *p, gsl_vector_complex *b, gsl_vector_complex *x, gsl_vector_complex *residual)
Complex version of mhc_linalg_LU_refine.

mhc_linalg_LU_invert(gsl_matrix *LU, gsl_permutation *p, gsl_matrix *inverse)
These functions compute the inverse of a matrix A from its LU decomposition (LU,p), storing the result in the matrix inverse. The inverse is computed by solving the system $Ax = b$ for each column of the identity matrix. It is preferable to avoid direct computation of the inverse whenever possible.

mhc_linalg_complex_LU_invert(gsl_matrix_complex *LU, gsl_permutation *p, gsl_matrix_complex *inverse)
Complex version of mhc_linalg_LU_invert.

mhc_linalg_QR_decomp(gsl_matrix *A, gsl_vector *tau)
This function factorizes the M-by-N matrix A into the QR decomposition A = Q R. On output the diagonal and upper triangular part of the input matrix contain the matrix R. The vector tau and the columns of the lower triangular part of the matrix A contain the Householder coefficients and Householder vectors which encode the orthogonal matrix Q. The vector tau must be of length $k = \min(M, N)$. The matrix Q is related to these components by, $Q = Q_k ... Q_2 Q_1$ where $Q_i = I - \tau_i v_i v_i^T$ and v_i is the Householder vector $v_i = (0, ..., 1, A(i + 1, i), A(i + 2, i), ..., A(m, i))$. This is the same storage scheme as used by LAPACK. The algorithm used to perform the decomposition is Householder QR (Golub & Van Loan, Matrix Computations, Algorithm 5.2.1).

mhc_linalg_QR_solve(gsl_matrix *QR, gsl_vector *tau, gsl_vector *b, gsl_vector *x)
This function solves the system $Ax = b$ using the QR decomposition of A into (QR, tau) given by gsl_linalg_QR_decomp.

mhc_linalg_QR_svx(gsl_matrix *QR, gsl_vector *tau, gsl_vector *x)
This function solves the system $Ax = b$ in-place using the QR decomposition of A into (QR,tau) given by gsl_linalg_QR_decomp. On input x should contain the right-hand side b, which is replaced by the solution on output.

mhc_linalg_QR_lssolve(gsl_matrix *QR, gsl_vector *tau, gsl_vector *b, gsl_vector *x, gsl_vector *residual)
This function finds the least squares solutionto the overdetermined system $Ax = b$ where the matrix A has more rows than columns. The least squares solution minimizes the Euclidean norm of the residual, $||Ax - b||$. The routine uses the QR decomposition of A into (QR, tau) given by gsl_linalg_QR_decomp. The solution is returned in x. The residual is computed as a by-product and stored in residual.

mhc_linalg_QR_QTvec(gsl_matrix *QR, gsl_vector *tau, gsl_vector *v)
This function applies the matrix Q^T encoded in the decomposition (QR,tau) to the vector v, storing the result $Q^T v$ in v. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q^T.

mhc_linalg_QR_Qvec(gsl_matrix *QR, gsl_vector *tau, gsl_vector *v)
This function applies the matrix Q encoded in the decomposition (QR,tau) to the vector v, storing the result Qv in v. The matrix multiplication is carried out directly using the encoding of the Householder vectors without needing to form the full matrix Q.

mhc_linalg_QR_Rsolve(gsl_matrix *QR, gsl_vector *b, gsl_vector *x)
This function solves the triangular system $Rx = b$ for x. It may be useful if the product $b' = Q^T$ b has already been computed using gsl_linalg_QR_QTvec.

mhc_linalg_QR_Rsvx(gsl_matrix *QR, gsl_vector *x)
This function solves the triangular system $Rx = b$ for x in-place. On input x should contain the right-hand side b and is replaced by the solution on output. This function may be useful if the product $b' = Q^T b$ has already been computed using gsl_linalg_QR_QTvec.

mhc_linalg_QR_unpack(gsl_matrix *QR, gsl_vector *tau, gsl_matrix *Q, gsl_matrix *R)
This function unpacks the encoded QR decomposition (QR,tau) into the matrices Q and R, where Q is M-by-M and R is M-by-N.

mhc_linalg_QR_QRsolve(gsl_matrix *Q, gsl_matrix *R, gsl_vector *b, gsl_vector *x)
This function solves the system $Rx = Q^T b$ for x. It can be used when the QR decomposition of a matrix is available in unpacked form as (Q,R).

mhc_linalg_QR_update(gsl_matrix *Q, gsl_matrix *R, gsl_vector *w, gsl_vector *v)
This function performs a rank-1 update wv^T of the QR decomposition (Q, R). The update is given by $Q'R' = QR + wv^T$ where the output matrices Q' and R' are also orthogonal and right triangular. Note that w is destroyed by the update.

mhc_linalg_R_solve(gsl_matrix *R, gsl_vector *b, gsl_vector *x)
This function solves the triangular system $Rx = b$ for the N-by-N matrix R.

mhc_linalg_R_svx(gsl_matrix *R, gsl_vector *x)
This function solvesthe triangular system $Rx = b$ in-place. On input x should contain the right-hand side b, which is replaced by the solution on output.

A.9 Eigenvectors and Eigenvalues

mhc_eigen_herm(gsl_matrix_complex *A, gsl_vector *eval, gsl_eigen_herm_workspace *w)

This function computes the eigenvalues of the complex hermitian matrix A. Additional workspace of the appropriate size must be provided in w. The diagonal and lower triangular part of A are destroyed during the computation, but the strict upper triangular part is not referenced. The imaginary parts of the diagonal are assumed to be zero and are not referenced. The eigenvalues are stored in the vector eval and are unordered.

mhc_eigen_symmv(gsl_matrix *A, gsl_vector *eval, gsl_matrix *evec, gsl_eigen_symmv_workspace *w)
This function computes the eigenvalues and eigenvectors of the real symmetric matrix A. Additional workspace of the appropriate size must be provided in w. The diagonal and lower triangular part of A are destroyed during the computation, but the strict upper triangular part is not referenced. The eigenvalues are stored in the vector eval and are unordered. The corresponding eigenvectors are stored in the columns of the matrix evec. For example, the eigenvector in the first column corresponds to the first eigenvalue. The eigenvectors are guaranteed to be mutually orthogonal and normalised to unit magnitude.

mhc_eigen_hermv(gsl_matrix_complex *A, gsl_vector *eval, gsl_matrix_complex *evec, gsl_eigen_hermv_workspace *w)
This function computes the eigenvalues and eigenvectors of the complex hermitian matrix A. Additional workspace of the appropriate size must be provided in w. The diagonal and lower triangular part of A are destroyed during the computation, but the strict upper triangular part is not referenced. The imaginary parts of the diagonal are assumed to be zero and are not referenced. The eigenvalues are stored in the vector eval and are unordered. The corresponding complex eigenvectors are stored in the columns of the matrix evec. For example, the eigenvector in the first column corresponds to the first eigenvalue. The eigenvectors are guaranteed to be mutually orthogonal and normalised to unit magnitude.

A.10 Fast Fourier Transforms

mhc_fft_complex_forward(gsl_complex_packed_array data[], size_t *stride, size_t *n, gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)
These function computes forward FFTs of length n with stride stride, on the packed complex array *data*, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length n. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing the trigonometric lookup tables and a workspace work.

mhc_fft_complex_transform(gsl_complex_packed_array data[], size_t *stride, size_t *n, gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work, gsl_fft_direction *sign)
These function computes transform FFTs of length n with stride *stride*, on the packed complex array data, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length n. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing the trigonometric lookup tables and a workspace work.

mhc_fft_complex_backward(gsl_complex_packed_array data[], size_t *stride, size_t *n, gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)
These function computes backward FFTs of length n with stride *stride*, on the packed complex array *data*, using a mixed radix decimation-in-frequency algorithm. There is no restriction on

the length n. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing the trigonometric lookup tables and a workspace work.

mhc_fft_complex_inverse(gsl_complex_packed_array data[], size_t *stride, size_t *n, gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)
These function computes inverse FFTs of length n with stride *stride*, on the packed complex array *data*, using a mixed radix decimation-in-frequency algorithm. There is no restriction on the length n. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing the trigonometric lookup tables and a workspace work.

mhc_fft_complex_transform(gsl_complex_packed_array data[], size_t *stride, *stride, size_t *n, gsl_fft_complex_wavetable *wavetable, gsl_fft_complex_workspace *work)
This function computes the FFT of *data*, a real array of length n, using a mixed radix decimation-in-frequency algorithm. For gsl_fft_real_transform data is an array of time-ordered real data. There is no restriction on the length n. Efficient modules are provided for subtransforms of length 2, 3, 4 and 5. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing trigonometric lookup tables and a workspace work.

mhc_fft_real_transform(double data[], size_t *stride, size_t *n, gsl_fft_real_wavetable *wavetable, gsl_fft_real_workspace *work)
This function computes the FFT of *data*, a half-complex array of length n, using a mixed radix decimation-in-frequency algorithm. For gsl_fft_halfcomplex_transform data contains fourier coefficients in the half-complex ordering described above. There is no restriction on the length n. Efficient modules are provided for subtransforms of length 2, 3, 4 and 5. Any remaining factors are computed with a slow, $O(n^2)$, general-n module. The caller must supply a wavetable containing trigonometric lookup tables and a workspace work.

A.11 Numerical Integration

mhc_integration_qags(gsl_function *f, double *a, double *b, double *epsabs, double *epsrel, size_t *limit, gsl_integration_workspace *workspace, double *result, double *abserr)
This function applies the Gauss-Kronrod 21-point integration rule adaptively until an estimate of the integral of f over (a,b) is achieved within the desired absolute and relative error limits, epsabs and epsrel. The results are extrapolated using the epsilon-algorithm, which accelerates the convergence of the integral in the presence of discontinuities and integrable singularities. The function returns the final approximation from the extrapolation, result, and an estimate of the absolute error, abserr. The subintervals and their results are stored in the memory provided by workspace. The maximum number of subintervals is given by limit, which may not exceed the allocated size of the workspace.

A.12 Statistics

mhc_stats_mean(double data[], size_t *stride, size_t *n, double *mean)
This function computes the arithmetic mean of data, a dataset of length n with stride stride. The result is returned in mean. The arithmetic mean, or sample mean, is denoted by $\hat{\mu}$ and defined as, $\hat{\mu} = (1/N) \sum x_i$ where x_i are the elements of the dataset data. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N.

mhc_stats_variance(double data[], size_t *stride, size_t *n, double *variance)
This function computes the estimated, or sample, variance of data, a dataset of length n with stride stride. The result is returned in variance. The estimated variance is denoted by $\hat{\sigma}^2$ and is defined by, $\hat{\sigma}^2 = (1/(N-1)) \sum (x_i - \hat{\mu})^2$ where x_i are the elements of the dataset data. Note that the normalization factor of $1/(N-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/N$. This function computes the mean via a call to gsl_stats_mean. If you have already computed the mean then you can pass it directly to gsl_stats_variance_m.

mhc_stats_variance_m(double data[], size_t *stride, size_t *n, double *mean, double *variance)
This function is the same as mhc_stats_variance except that the mean is passed in as mean instead of being computed.

mhc_stats_sd(double data[], size_t *stride, size_t *n, double *sd)
The standard deviation is defined as the square root of the variance. This function returns the square root of the corresponding variance functions above in sd.

mhc_stats_sd_m(double data[], size_t *stride, size_t *n, double *mean, double *sd_m)
This function is the same as mhc_stats_sd except that the mean is passed in as mean instead of being computed.

mhc_stats_covariance(double data1[], size_t *stride1, double data2[], size_t *stride2, size_t *n, double *covar)
This function computes the covariance of the datasets data1 and data2 which must both be of the same length n. $covar = (1/(n-1)) \sum_{i=1}^{n} (x\_i - \hat{x})(y_i - \hat{y})$

Appendix B

SAMPLE CONFIGURATION FILES

The following are samples of all of the different configuration files that are used by the mHC scheduler and/or the mHC Simulator. All of the configuration files are based on standard XML and can automatically be generated by the mHC simulator.

B.1 Device Configuration

The device configuration file is used by both the mHC scheduler and the mHC Simulator and determines what devices are available in the system. The following sample shows a definition for a host processor and a vector processor, each supporting only the mhc_matrix_scale operation to varying degrees.

```
<?xml version="1.0"?>
<mHCDeviceConfig>
  <Device>
    <ID>0</ID>
    <Name>Host</Name>
    <Description>The host processor</Description>
    <BusName>Local</BusName>
    <BusID>0</BusID>
    <APISupport>
      <Function>
        <ID>5</ID>
        <Name>mhc_matrix_add_constant</Name>
        <TimingType>0</TimingType>
        <TimingValue>1</TimingValue>
        <Speedup>1</Speedup>
        <CompletionTime>0.006</CompletionTime>
      </Function>
      <Function>
        <ID>4</ID>
        <Name>mhc_matrix_scale</Name>
        <TimingType>0</TimingType>
        <TimingValue>1</TimingValue>
        <Speedup>1</Speedup>
        <CompletionTime>0.006</CompletionTime>
      </Function>
    </APISupport>
  </Device>
  <Device>
    <ID>1</ID>
    <Name>Vector1</Name>
    <Description></Description>
    <BusName>PCI</BusName>
    <BusID>1</BusID>
    <APISupport>
      <Function>
        <ID>4</ID>
        <Name>mhc_matrix_scale</Name>
        <TimingType>0</TimingType>
        <TimingValue>1</TimingValue>
        <Speedup>2</Speedup>
        <CompletionTime>0.003</CompletionTime>
      </Function>
    </APISupport>
  </Device>
```

B.2 Bus Configuration

The bus configuration file is used by both the mHC scheduler and the mHC Simulator and determines what buses are available in the system. The following sample shows a definition for a local bus and a PCI bus.

```
<?xml version="1.0"?>
<mHCBusConfig>
  <Bus>
    <ID>0</ID>
    <Name>Local</Name>
    <Description>The local bus.</Description>
    <InitTime>0</InitTime>
    <Overhead>0</Overhead>
    <TransferTime>0</TransferTime>
  </Bus>
  <Bus>
    <ID>1</ID>
    <Name>PCI</Name>
    <Description>A standard PCI bus.</Description>
    <InitTime>50</InitTime>
    <Overhead>0.01</Overhead>
    <TransferTime>0.005</TransferTime>
  </Bus>
</mHCBusConfig>
```

Appendix C

SAMPLE MHC APPLICATION

The following is a sample application to demonstrate some of the constructs required to use the microHeterogeneous API. This application performs a simple vector scaling on a single vector. It demonstrates the correct way to initialize and finalize the mHC API. This program can be compiled and linked by using:

```
gcc -lmhc -o sample sample.c
```

Note that this assumes that mHC_api.h is located in the include path and libmhc.so.1.0.0 is located in the lib path. If this is not the case, simply use the $-I$ and $-L$ arguments of gcc to point to the files directly.

The easiest way to run the application to make sure that the mHC library and header files are installed correctly is to use:

```
sample -s -1
```

which runs the application sequentially and thus does not require the normal device and bus configuration files. If there are no errors during compilation or execution, then the mHC API has been installed correctly.

```
#include <stdio.h>
#include <stdlib.h>
#include "mHC_api.h"

int main( int argc, char **argv )
{
    int i;
    gsl_vector *x = gsl_vector_alloc( 100 );
    double scale = 5;

    /* initialize the mHC api by passing it the argument list */
    if( mhc_initialize( argv ) != MHC_SUCCESS )
        abort();

    /* initialize the vector */
    for( i = 0; i < 100; i++ )
        gsl_vector_set( x, i, i + .75 );

    /* make a simple mhc call and wait for it to finish*/
    mhc_vector_scale( x, &scale );
    mhc_join();

    /* free the vector that we created */
    gsl_vector_free( x );

    /* cleanup the mHC api, must be done to free memory and close log */
    if( mhc_finalize() != MHC_SUCCESS )
        abort();

    return 0;
} /* main */
```

Appendix D

SAMPLE MHC SIMULATOR REPORT

A few samples of reports that were automatically generated by the mHC Simulator can be found in the */generated_reports/* folder on the CD_ROM. These reports are HTML based and should be viewed by pointing a web browser to */generated_reports/index.html*.

Appendix E

MHC FRAMEWORK SOURCE CODE

The source code for the microHeterogeneous framework is located in the */source/mHC_scheduler/* directory on the CD_ROM.

Appendix F

MHC SIMULATOR SOURCE CODE

The source code for the microHeterogeneous Simulator is located in the */source/mHC_simulator/* directory on the CD_ROM.

BIBLIOGRAPHY

[1] Argonne National Laboratory. *MPICH - A Portable Implementation of MPI.* http://www-unix.mcs.anl.gov/mpi/mpich/, 2002.

[2] Armstrong, R., D. Hensgen and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *7th IEEE Heterogeneous Computing Workshop (HCW'98)*, pages 79–87, March 1998.

[3] Barbosa, D. M., João P. Kitajima and W. Meira Jr. Parallelizing MPEG Video Encoding using Multiprocessors. In *Proceedings of the XII Brazilian Symposium on Computer Graphics and Image Processing*, October 1999.

[4] Bokhari, S. H. On the mapping problem. In *IEEE Transactions on Computers*, volume C-30, pages 207–214, 1981.

[5] Braun, T., H. Siegel, N. Beck, L. Boloni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen and R. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *8th IEEE Heterogeneous Computing Workshop (HCW'99)*, pages 15–29, April 1999.

[6] Catalina Research. *Pegasus-2 - High Speed Vector Processing for the PCI Bus.* http://www.catalinaresearch.com/_upload/pdf/pegasus2_ds.pdf, 2002.

[7] Chow, K. and B. Liu. On mapping signal processing algorithms to a heterogeneous multiprocessor system. In *ICASSP 91*, page 1585:1588, May 1991.

[8] Dillon, E., C. Gamboa Dos Santos, J. Guyard. Homogeneous and Heterogeneous Networks of Workstations: Message Passing Overhead.

http://www.osl.iu.edu/download/mpidc95/papers/html/dillon/index.html, June 1995.

[9] El-Rewini, H. and T.G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. In *Journal of Parallel and Distributed Computing*, volume 9, pages 138–153, 1990.

[10] Freund, R., M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. Lima, F. Mirabile, L. Moore, B. Rust and H. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet. In *7th IEEE Heterogeneous Computing Workshop (HCW'98)*, pages 184–199, March 1998.

[11] Galassi, M., J. Davies and J. Theiler. In *GNU Scientific Library Reference Manual*. Network Theory Ltd, December 2001.

[12] GNU. http://www.gnu.org/copyleft/gpl.html, 2002.

[13] Hughes Research Laboratory. *Vector, Signal, and Image Processing Library (VSIPL)*. http://www.vsipl.org, 2002.

[14] Iverson, M., F. Ozguner, G. Follen. Parallelizing Existing Applications in a Distributed Heterogeneous Environment. In *Heterogeneous Computing Workshop (HCW'95)*, pages 93–100, 1995.

[15] Khokhar, A., V. K. Prasanna, M. Shaaban and C. Wang. Heterogeneous Supercomputing: Problems and Issues. In *Workshop on Heterogeneous Processing (WHP'1992)*, pages 3–12, 1992.

[16] Myricom. *Myrinet Performance Measurements*. http://www.myri.com/myrinet/performance, July 2002.

[17] Netlib. *BLAS - Basic Linear Algebra Subprograms*. http://www.netlib.org/blas/, 2002.

[18] Netlib. *PVM - Paralllel Virtual Machine*. http://www.netlib.org/pvm3/book/node17.html, 2002.

[19] OpenGL. http://www.opengl.org/, 2002.

[20] Sih, G.C. and E.A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. In *IEEE Trans. Parallel and Distributed Systems*, volume 4, pages 175–186, February 1993.

[21] Tao, L., B. Narahari and Y. Zhao. Heuristics for Mapping Parallel Computations to Heterogeneous Parallel Architectures. In *Workshop on Heterogeneous Processing (WHP'1993)*, pages 36–41, 1993.

[22] Texas Memory Systems, Inc. *XP-15 DSP Accelerator Card*. http://www.texmemsys.com/files/f000090.pdf, 2002.

[23] Topcuoglu, H., S. Hariri and M. Wu. Task Matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. In *Journal of Parallel and Distributed Computing*, volume 13-3, pages 260–274, March 1992.

[24] University of Hawaii. *Data Bus Speed Comparison Table*. http://www.hawaii.edu/infotech/busspeeds.html, October 2001.

[25] Wang, L., H. J. Siegle, V. P. Roychowdhury and A. A. Maciejewski. Task Matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. In *Journal of Parallel and Distributed Computing*, volume 47-1, pages 1–15, November 1997.

[26] Wang, M. Augmenting the Optimal Selection Theory for Superconcurrency. In *Workshop on Heterogeneous Processing (WHP'1992)*, pages 13–21, 1992.

[27] Weems, C. Image Understanding: A Driving Application for Reseach in Heterogeneous Parallel Processing. In *Workshop on Heterogeneous Processing (WHP'1993)*, pages 119–126, 1993.

[28] Wu, M. and W. Shu. A high-performance mapping algorithm for heterogeneous computing systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[29] Wu, M., W. Shu and H. Zhang. Segmented min-min: a static mapping algorithm for meta-tasks on heterogeneous computing systems. In *9th IEEE Heterogeneous Computing Workshop (HCW'00)*, pages 375–385, March 2000.