Parallelization of Finite Element Analysis Codes Using Heterogeneous Distributed Computing

NASA Grant: NAG3-1440

Final Report

Start Date: January 15, 1993
End Date: March 31, 1996

Submitted by:

Füsun Özgüner
Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210
ozguner@ee.eng.ohio-state.edu

RF Project No. 760492/727127

Contents

List of Figures

1. Introduction

Performance gains in computer design are quickly consumed as users seek to analyze larger problems to a higher degree of accuracy. Innovative computational methods, such as parallel and distributed computing, seek to multiply the power of existing hardware technology to satisfy the computational demands of large applications. In the early stages of this project, experiments were performed using two large, coarse-grained applications, CSTEM and METCAN. These applications were parallelized on an Intel iPSC/860 hypercube. It was found that the overall speedup was very low, due to large, inherently sequential code segments present in the applications. The overall execution time T_{par} of the application is dependent on these sequential segments. If these segments make up a significant fraction of the overall code, the application will have a poor speedup measure. This relationship is governed by Amdahl's law, which states that if f is the fraction of code that is sequential (cannot be parallelized) then the actual speedup is bounded by the equation:

$$S_{max} \leq \frac{1}{f + (1 - f)/N} \tag{1}$$

In addition to the problems with sequential code segments, the programming process was extremely difficult, since the applications were not written to support parallel computing. From these experiments, it was evident that another parallel computing paradigm was needed to effectively increase the performance of this type of application: heterogeneous computing.

A typical application will contain a number of code segments. Each of these segments will be best suited to a different type of computer architecture. Therefore, an effective way of increasing the performance of an application is to break the application into fragments, and execute each fragment on the best suited architecture, in parallel wherever possible. This technique is known as heterogeneous computing. Heterogeneous computing is particularly well suited to large coarse grained applications, like CSTEM. Heterogeneous computing can be formally defined as "the 'tuned' use of diverse processing hardware to meet distinct computational needs, in which ... code portions are executed using processing approaches that maximize overall performance" [15]. This definition stems from the observation that most high-performance computers are optimized for a particular type of computation, and often perform very poorly when executing other types of code. Therefore, the performance bottleneck tends to be in the portions of the code that do not execute efficiently. To eliminate this bottleneck, we can use heterogeneous computing to execute each code fragment of an application on the best suited architecture. In this way, heterogeneous computing is used as a means of increasing the performance of an application beyond the level it

can achieve on any single machine. The specific heterogeneous environment considered in this project is a loosely coupled set of independent machines. The use type of environment is known as "heterogeneous distributed computing." Since nearly all current heterogeneous systems fall into this category, the name "heterogeneous computing" will be used interchangeably.

The effectiveness of heterogeneous computing (or any kind of parallel computing) is determined by the choice of which processor should execute each task of the application. Typically, this is made to satisfy some set of cost functions. The process of determining such an assignment is called the matching and scheduling problem. The solution of the matching and scheduling problem is non-trivial, and the problem is NP-hard [32]. Therefore, heuristic solutions are commonly used to obtain solutions. In the case of conventional, homogeneous parallel computing, the analog to the matching and scheduling problem is known as the *mapping and scheduling problem.* To simplify notation, we will refer to both of these problems as "the matching and scheduling problem." The difference in nomenclature is due to the heterogeneity of the target machines, where individual tasks and machines are "matched" as opposed to a task being "mapped" onto a homogeneous set of processors.

In order for a matching and scheduling algorithm to make an effective scheduling decision, an accurate set of estimates of the execution time of the task on each potential machine is needed. Since the execution time of a task depends upon the input data, this problem is rather difficult. In a homogeneous environment, this is a fairly simple task, since all tasks will perform the same on each processor. However, in a heterogeneous environment, each task will behave differently depending upon the machine on which it is run. This feature of heterogeneous environments makes the process of obtaining estimates of the execution time (called the execution time estimation problem) very difficult.

The goal of this project was to develop practical techniques for heterogeneous computing, through experiments with a large, coupled finite element application. The application chosen, CSTEM, is a good candidate for use with heterogeneous computing, due to its size and structure. Prior attempts to parallelize it using conventional parallel processors were both time consuming and impractical, as discussed above. Therefore, we chose to use it to show the advantages of heterogeneous computing for large scientific applications. To effectively use CSTEM with heterogeneous computing, we developed a heuristic method for solving the matching and scheduling problem in a heterogeneous environment. Furthermore, as we indicated above, any matching and scheduling algorithm requires a set of execution time estimates. Therefore, we have developed a statistical method for predicting the execution time of a task based upon past values.

The remainder of this report is organized as follows. Section 2 will introduce CSTEM and describe HeNCE, the tool used to create the heterogeneous version of CSTEM. In addition, Section 2 will also provide detailed explanation of the software

engineering decisions made in creating heterogeneous CSTEM. Section 3 will address the matching and scheduling problem, by presenting a new heuristic called the *LMT Algorithm*. Since the matching and scheduling algorithm can only make good decisions when it has accurate execution time estimates, Section 4 will present a statistical scheme for estimating the execution time of tasks using past observations. Finally, Section 5 will offer some conclusions from the results obtained in this project.

2. Heterogeneous CSTEM

CSTEM, an acronym for Coupled Structural/Thermal/Electromagnetic Analysis/Tailoring of Graded Composite Structures, is a finite element-based computer program developed for the NASA Lewis Research Center [16]. As its name implies, CSTEM analyzes and optimizes the performance of composite structures using a variety of dissimilar analysis modules, including a structural analysis module, a thermal analysis module, an electromagnetic absorption analysis module, and an acoustic analysis module. Large, coupled structures codes, like CSTEM, have huge demands for computational resources. CSTEM, for example, consists of approximately 81,000 lines of FORTRAN code, and requires several minutes of CPU time even for small, trivial problems.

To increase the performance of CSTEM, an initial attempt was made to parallelize a significant portion of the code on the Intel iPSC/860 hypercube. It was quickly apparent that this approach was impractical, due to the overall size of the source code, the large amount of memory required by CSTEM, and the limitations inherent in the FORTRAN programming language. The data parallel programming style of the Intel hypercube and other distributed memory parallel processors is still a valid method for obtaining a significant speedup in the performance of CSTEM, however. The data parallel approach needs to be used in conjunction with some other method that can help alleviate the problems caused by the size of the code and the memory requirements of the program.

These problems can be minimized through the use of heterogeneous distributed computing. First, since the code is divided into a number of independent tasks, the overall size of the source code and memory requirements of each individual task are greatly reduced (memory use is reduced because FORTRAN only uses static data allocation). Second, the total amount of parallelization that must be performed is reduced, since the tasks that do not make a significant contribution to the total execution time and the tasks that are not part of the critical execution path do not contribute to the overall execution time. Therefore, these tasks do not need to be placed on a parallel processor in order to increase the overall performance of the application. Finally, tasks that are ill suited to the distributed memory parallel architecture can be run on another architecture that is better suited to that type of computation.

Clearly, heterogeneous computing simplifies the subsequent parallelization of CSTEM on a distributed memory multiprocessor. But, before the heterogeneous version is presented, some background information on HeNCE, the tool used to create the heterogeneous CSTEM will be presented, along with details of PVM, the heterogeneous message passing library used by HeNCE.

2.1 HeNCE

HeNCE, an acronym for Heterogeneous Network Computing Environment, is an automated tool for the development of heterogeneous applications developed at Oak Ridge National Laboratories. Using HeNCE, a programmer can quickly write a heterogeneous application, since HeNCE eliminates the tedious task of writing the code used to maintain and coordinate a set of processes running on different machines and the code used to transfer data between the different processes. To create a heterogeneous application, the HeNCE programmer only needs to provide a set of C or FORTRAN function calls and a data dependency graph. HeNCE also provides a graphical interface, called *htool*, for creating HeNCE applications and for graphical performance monitoring. HeNCE is built upon the PVM (Parallel Virtual Machine) message passing libraries, also developed at Oak Ridge. PVM provides a set of message passing, synchronization, and data conversion utilities allowing a heterogeneous set of networked machines to communicate with each other in a manner similar to distributed memory multiprocessors.

There are several steps required to create and run a program using HeNCE [3]:

1. Create a program graph, specifying a function call and parameters for each node.
2. Write sequential code for each node, based upon the function call specified above.
3. Provide the names of the machines that are to execute the code.
4. Input estimated computation costs for each subroutine/machine pair.
5. Automatically generate wrapper code with necessary PVM function calls.
6. Automatically build makefiles, and use these to build the executable files.
7. Execute the program and trace the results.

All of these steps can be initiated from *htool*. An important feature of HeNCE is that there is no requirement that the source code for a node's function call be the same on each different architecture. For example, HeNCE can choose, at run time, to use either a vectorizable algorithm or a parallelizable algorithm, depending only on the machine to which the node is assigned. This allows the optimum algorithm to be used on each architecture.

2.1.1 The Program Graph

The *program graph* defines the data dependencies that exist between the functions comprising the HeNCE application. The program graph is a directed, acyclic graph (DAG), with the nodes of the graph representing the subroutine to be executed and the edges representing the data dependencies between the nodes. Please note that HeNCE uses an "upside-down" convention, where graphs flow from bottom to top. A sample program graph is shown in Figure 2.1. In addition to nodes that call functions, HeNCE also has a number of special purpose nodes which allow the graph to be dynamically reconfigured as the application executes. There are four types of special nodes, providing four different styles of control flow: loops, parallel loops (fan), conditional execution, and pipelined execution. Figure 2.2 shows how these nodes appear in *htool*. The loop nodes allow the set of nodes enclosed by the pair to be executed multiple times in a sequential manner. Fan nodes operate in the same way as loop nodes, but execute the set of nodes in parallel. Pipeline nodes send multiple sets of data through a set of nodes in a pipelined manner. Finally, conditional nodes evaluate an expression, and based upon the result, conditionally execute the enclosed nodes.

The execution of each node in the program graph has three phases [3]:

1. **Getting Parameters:** The node queries its ancestors for the data required for its computation. The node gets its data from the closest ancestor. For example, the node program first checks to see if its parent nodes have the necessary information, then its grandparent nodes, etc.

2. **Executing the Function Call:** The node calls its associated subroutine.

3. **Sending Parameters:** At this point, the node has finished its execution, and its children may be started. However, the node does not terminate. Instead, the node sleeps, waiting to provide any data required by its descendants when requested.

This three-phased execution process forces the program graph to be an acyclic graph. A node cannot send data to a descendant node and later receive data from that descendant without violating the execution process defined above. The node's execution process is controlled by a *node program*. The node program is specified using HeNCE's *node language*. A node program specifies the parameters held by a node, how to acquire them, and how to pass them to the node's subroutine. There are three types of parameters, input only, input-output, and output only. Input parameters are loaded from an ancestor node, and cannot be sent to a descendant. Input-output parameters are loaded from an ancestor, and may be sent to a descendant. Output parameters are created within the node, and may be sent to a descendant. A parameter can be specified to be "NEW", telling the node to create and initialize the

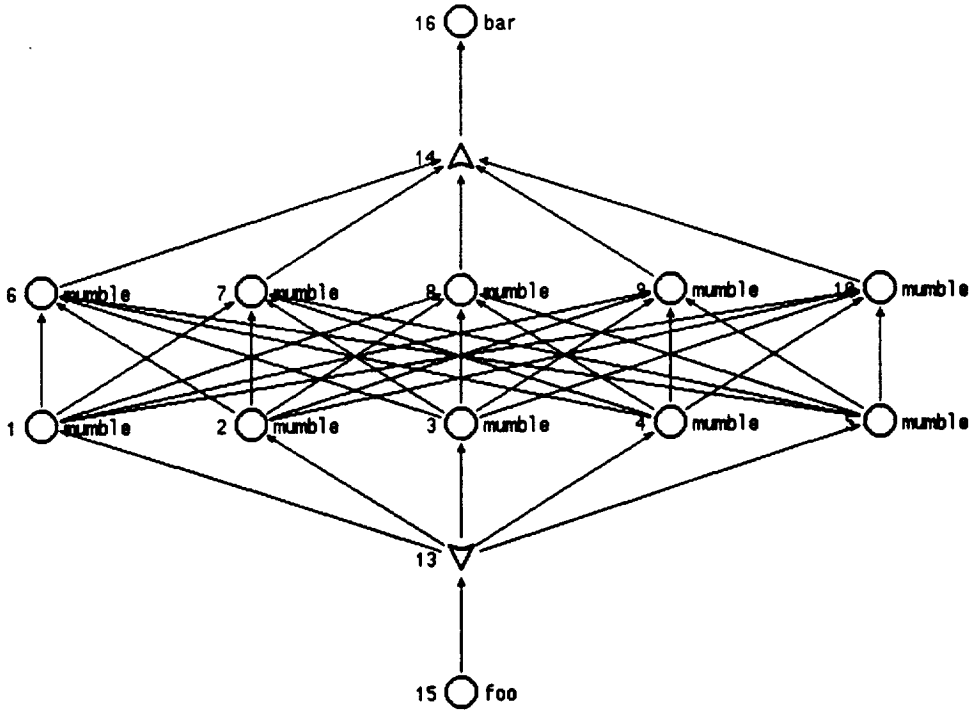


Figure 2.1: Sample HeNCE Program Graph

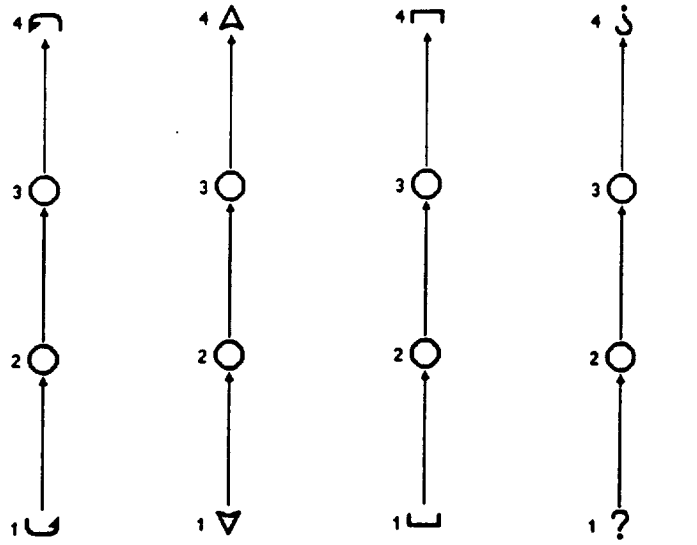


Figure 2.2: Special Purpose HeNCE Nodes.

parameter locally. Otherwise, the parameter is loaded from the node's ancestors. If the parameter does not exist on any ancestor, however, it is treated as if it were "NEW".

2.1.2 Costs Matrix

The *costs matrix* defines the suitability of a node to be executed on each machine used in the computation. An integer value is assigned to each node/machine pair, indicating the estimated cost of executing the node on that machine. A cost of zero indicates that a node cannot be assigned to that machine. HeNCE allocates nodes to machines using a simple method. HeNCE keeps track of the total cost it has assigned to each machine, and, when each node is ready to begin, HeNCE will assign that node to the machine such that the resulting total cost is minimized [4].

2.1.3 Performance Monitoring

HeNCE provides a graphical performance monitoring facility in *htool*, providing information on process timing, machine allocation, and communication overhead. It is capable of running in real time or as a post-mortem analysis. Figure 2.3 and Figure 2.4 show sample views of this facility.

2.2 PVM

As was stated above, HeNCE is built upon the PVM message passing library. PVM provides message passing and synchronization functions to user applications running on a wide variety of workstations and supercomputers. These routines allow a collection of networked machines to function as if it were a distributed memory parallel processor. Furthermore, there is no requirement that the communicating machines are homogeneous, making the development of true heterogeneous applications possible. PVM also incorporates automatic data conversion for communication between dissimilar machines. These facts, combined with the wide variety of machines supporting PVM, made PVM the ideal choice as the message passing mechanism for HeNCE [4].

2.2.1 The Implementation of PVM

Figure 2.5 shows a diagram detailing the implementation of PVM. On each machine included in the computation, a *PVM daemon* process must be present. This process serves as an interface between the PVM processes running on that machine and the network. PVM processes communicate with the daemon through a set of function calls which together form the *PVM library*. The PVM daemon and library are described in greater detail in [17]. The most significant aspect of this structure is to note that

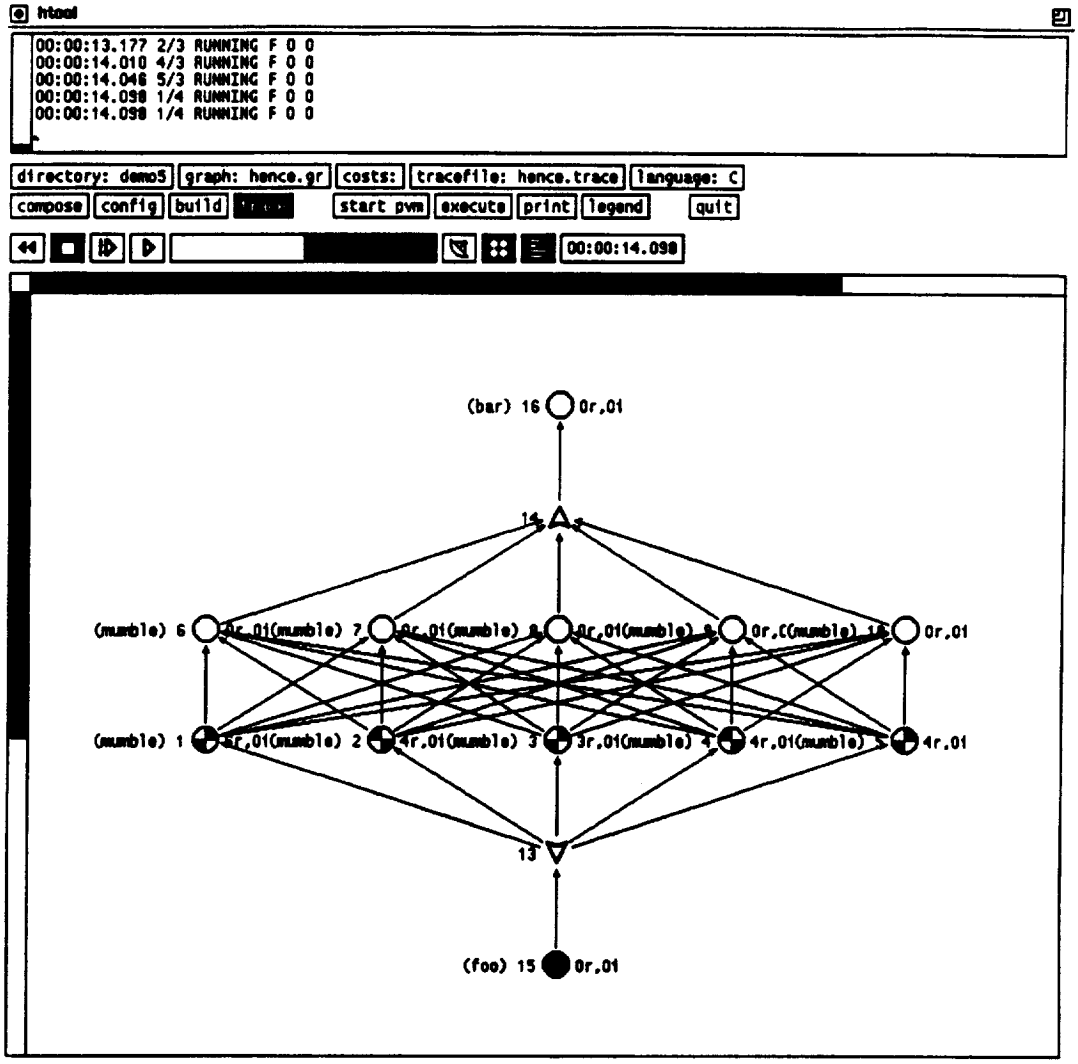


Figure 2.3: The Main Window of *htool's* Trace Mode

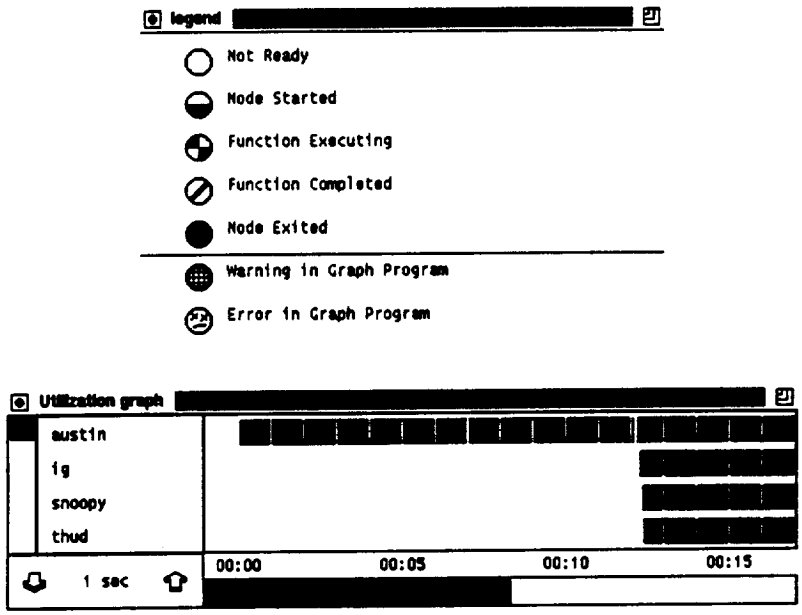


Figure 2.4: Other Windows from *htool's* Trace Mode

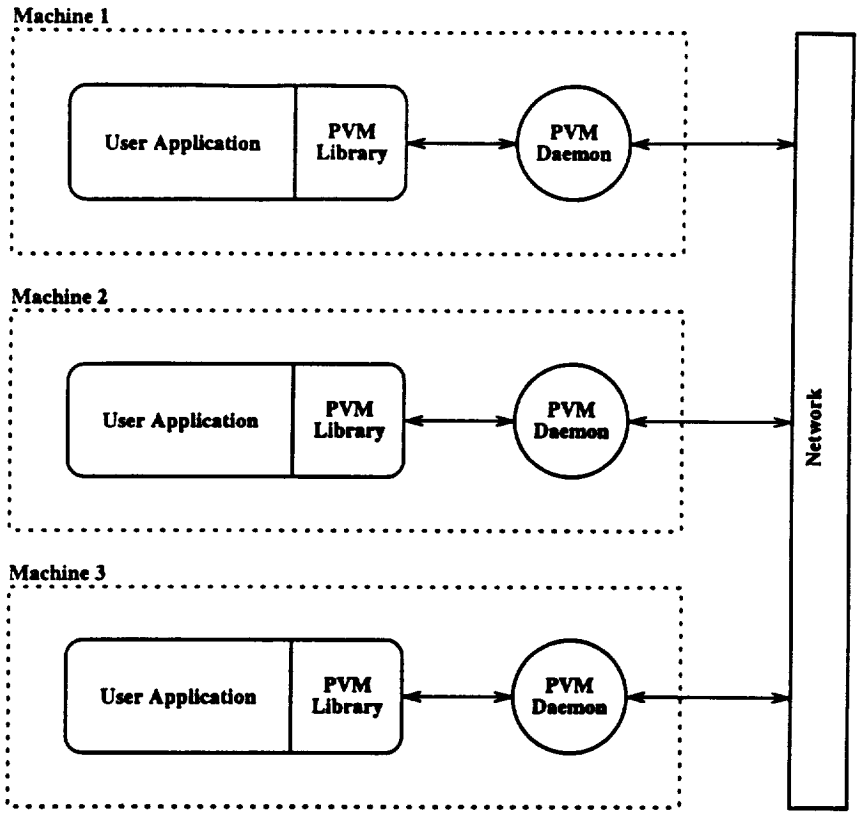


Figure 2.5: The Implementation of the PVM Interface

it causes PVM to have a relatively high overhead associated with each message. Each message sent is copied either five times, if the source and destination processes are on different machines, or four times, if both processes are on the same machine. The five copying steps are from the source process's data structure to the PVM library, PVM library to the daemon, over the network to the destination daemon, to the destination process's library, and finally to the destination process's data structures. Minimizing communication volume and frequency, therefore, are critical to designing a good PVM-based application.

2.2.2 Optimized PVM Versions

On machines with special high performance networking features, optimized versions of PVM have been constructed to help alleviate the high cost of sending PVM messages. For example, there is a version of PVM for the Intel iPSC/860 hypercube which translates PVM message passing calls into native Intel function calls, allowing PVM applications to take advantage of the network features available on the hypercube,

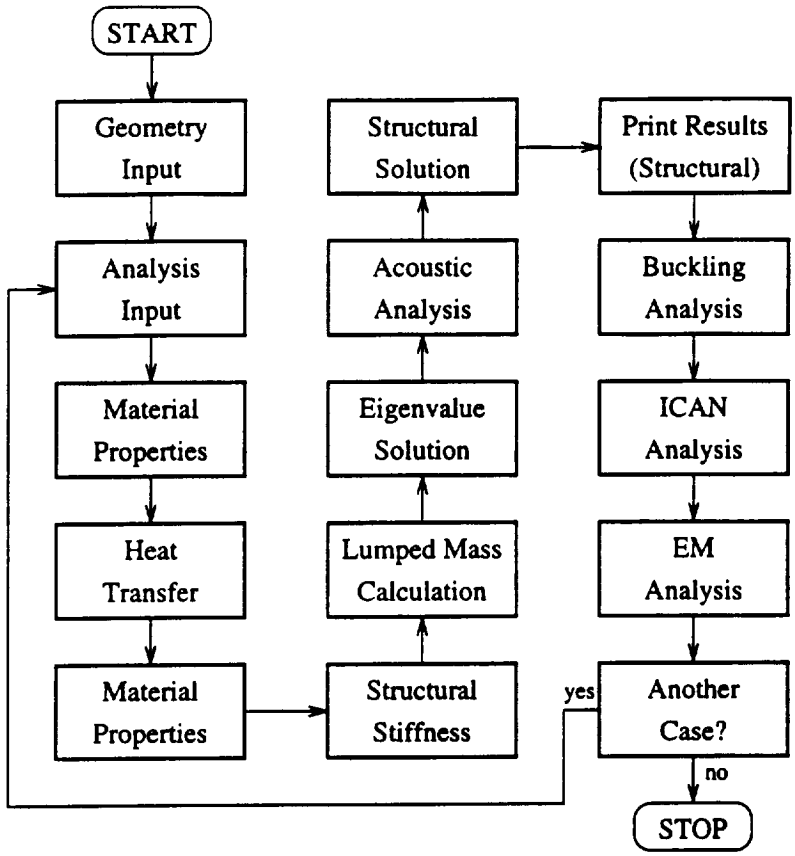


Figure 2.6: The Main Analysis Routine of CSTEM.

while still remaining portable between different machines supporting PVM. There also is an optimized version available on NASA Lewis Research Center's LACE cluster, a collection of 32 IBM RS/6000 workstations connected by a high performance network.

2.3 CSTEM

As mentioned in the beginning of this section, CSTEM combines a variety of multi-disciplinary analysis options together to provide a unified tool for the design and optimization of composite aircraft gas turbine engine (AGTE) components. CSTEM consists of a number of stand-alone finite element codes. These codes are coupled together by an iterative optimization routine. Figure 2.6 shows a flowchart detailing the main analysis routine. The optimization routine executes this routine multiple times, adjusting the design until it converges upon the desired solution.

The finite element method is a popular method of finding an approximate solution to a complex system. The popularity of the finite element method is due in part to its ability to be applied to a wide variety of systems in a broad range of different fields. For example, the finite element method has broad applications in the fields of mechanics, thermodynamics, electromagnetics, and fluid dynamics, to name just a few. The finite element method can be applied to both discrete and continuous systems; for discrete systems, the actual solution to the system is computed, while an approximate solution is calculated when applied to continuous systems. In the case of CSTEM, the system is a three-dimensional composite structure, and this system is solved for the variety of responses described above, including heat transfer, structural displacement under different loads, and electromagnetic absorption. Further details about the finite element method and CSTEM can be found in [2] and [16, 30], respectively.

2.4 The Heterogeneous Version of CSTEM

There are several important design issues that arose during the creation of the heterogeneous version of CSTEM. The most significant of these issues was to develop a set of criteria that could be used to split CSTEM into a number of independent code blocks. Once this issue was determined, issues of secondary importance could be established: including methods of passing data structures between blocks, sharing files between blocks, and keeping the data in these files coherent. This rest of this section will discuss how these issues were resolved, and will examine the structure of the heterogeneous version of CSTEM.

2.4.1 Code Division

As described above, CSTEM consists of a number of diverse analysis routines coupled together to form a single analysis tool. (See Figure 2.6 for the flowchart.) Almost all of these analysis routines are called from a single FORTRAN subroutine. Data is passed between these routines using three different means: as parameters of functions, in COMMON blocks, or in files. To complicate matters, these methods are often used interchangeably and inconsistently, largely because CSTEM derives a large portion of its code from existing applications.

For the initial version of heterogeneous CSTEM, it was decided, for the sake of simplicity, not to exploit any *data parallelism* in the code. When using data parallelism, the data is divided among several processors, with each processor executing the same code on its share of the data. Instead, CSTEM would exploit only *task parallelism*—where each processor receives all of the data, but the code is split into a series of heterogeneous subtasks, each assigned to a different processor, executing in parallel where possible. A number of factors need to be balanced when splitting

the code in this manner. Any split should attempt to maximize the number of tasks executing in parallel, minimize the communications volume between tasks, and split the code at function call boundaries, in order to minimize the number of modifications to existing code.

After weighing all of these parameters, CSTEM was divided into 15 separate tasks. Each of these tasks has its own separate source code, simplifying the process of porting individual tasks to different architectures. Although no data parallelism is used in this version, individual tasks that are in the critical execution path can be parallelized using data parallel techniques to increase the overall performance. Figure 2.7 shows a representation of the resulting HeNCE program graph. The numeric values within the nodes of the graph represent the approximate execution time, in seconds, upon a Sun workstation, excluding any communication costs. The edges between the nodes represent the precedence relationships between the nodes. These edges do not represent all of the nodes which communicate, however. Adding all of the communication edges would have made the graph unreadable. The names assigned to the nodes are arbitrary names uniquely identifying each task. As stated above, CSTEM was mostly split along function call boundaries, so the individual tasks do not necessarily correlate with the items in the flowchart shown in Figure 2.6. From the numeric values in the program graph, the tasks causing performance bottlenecks are clearly visible, making the tasks "xsnd()" and "xstiff()" definite candidates for parallelization in later versions. Now, with the code split into a set of individual tasks, the next section will examine how data is passed between those tasks.

2.4.2 Data Sharing

As stated above, data is passed between routines using three methods: as function parameters, in COMMON blocks, or in files. Passing data through function parameters is the simplest scheme to contend with. Each parameter is made into a HeNCE variable, which HeNCE will pass on demand between the different tasks. Data passed through COMMON blocks and files require more specialized handling, however. Since data held in a COMMON block cannot be passed as a function parameter, the data in the COMMON block must be copied into a series dummy variables. These variables are passed between tasks as function parameters, and the data is copied back into the appropriate COMMON block. This method, however, produces a large, unmanageable number of dummy variables. Fortunately, the FORTRAN EQUIVALENCE statement provides a manageable way to accomplish this task. The EQUIVALENCE statement allows two dissimilar data structures to occupy the same block of physical memory. Using this statement, the COMMON block can be made to share its memory with an array of integers. The data in this array can easily be copied into a dummy array and then be passed between tasks.

The final method of passing data between blocks is through files. CSTEM is

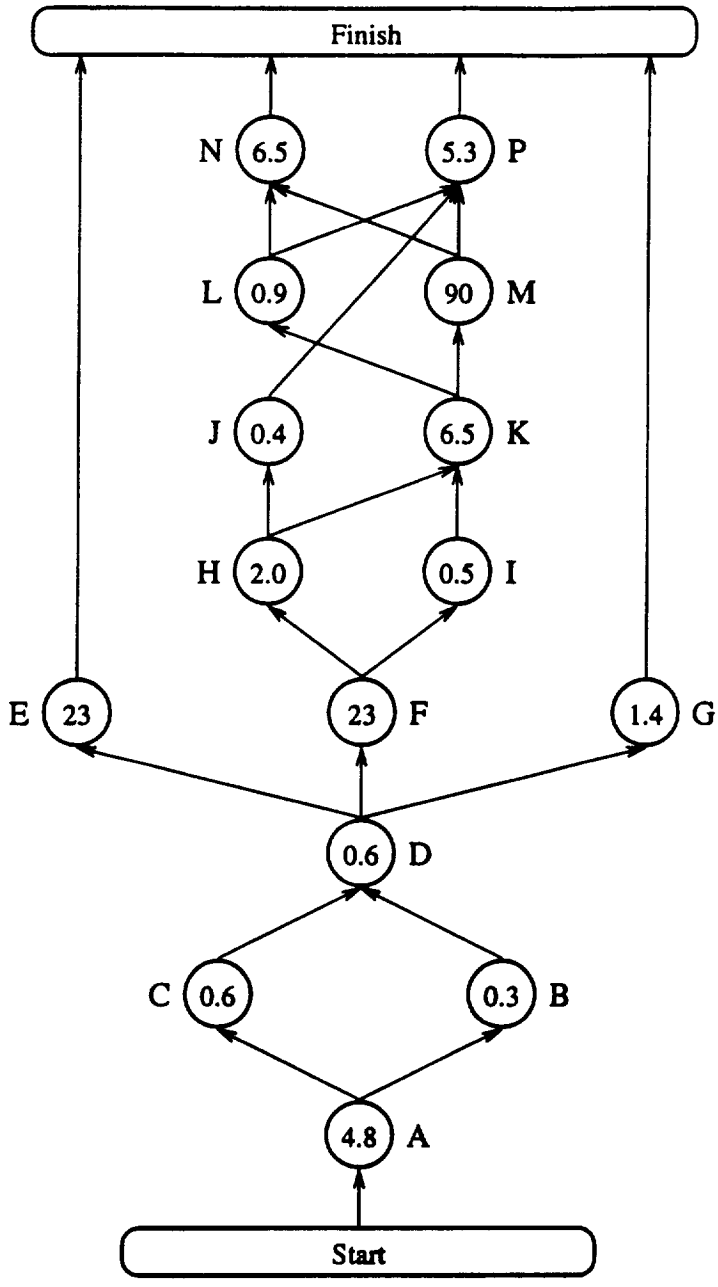


Figure 2.7: HeNCE Program Graph for Heterogeneous CSTEM

very dependent upon its file structure; it is almost constantly performing some file operation. CSTEM was designed to minimize the amount of physical memory used by keeping all unnecessary data stored in files.

At the present time, there is no coherent parallel file system available on the machines used by heterogeneous CSTEM. Therefore, two techniques have been used to simulate such a file system. The first is to use Sun's Network File System, also known as NFS. NFS allows files stored on a remote file server to be accessed locally by multiple machines, each as if the file was on a local disk. Thus, NFS allows the tasks of heterogeneous CSTEM to simultaneously read a single file. No writing may be done to a file while another task has that file open for reading, since data corruption may occur.

Although most workstations support NFS, not all machines are linked by a single NFS volume. NFS cannot be configured by a user, limiting the use of NFS to machines configured by the system administrator to share a file system. For machines not connected by NFS, another method of file sharing must be used, based upon the remote copy network service, also known as *rcp*. *Rcp* is a network service allowing files to be copied between different machines. To use *rcp* with heterogeneous CSTEM, one machine, preferable the machine either running the most tasks or having access to the NFS volume used by most of the tasks, would hold all of the files. Tasks not able to access files would use *rcp* to copy the needed files to a local drive before beginning execution, and would return modified files to the host machine upon completion.

Clearly, transferring files via NFS and *rcp* have significant drawbacks, but until a full-featured parallel file system is available for distributed workstations, they are the only options available to support file operations across heterogeneous machines.

2.5 Summary

Clearly, for large applications, there is a distinct advantage to using heterogeneous computing to exploit parallelism and simplify the programming process. HeNCE provides a convenient tool for quickly creating heterogeneous applications, and, therefore, was used to create heterogeneous CSTEM. Now, an efficient matching and scheduling algorithm is needed to allocate the individual tasks to a set of heterogeneous machines. The next section will examine the proposed matching and scheduling technique developed in this project.

3. Matching and Scheduling

As stated above, the most difficult problem associated with heterogeneous distributed computing, as well as parallel computing in general, is the matching and scheduling problem. This problem assigns the individual code fragments, or tasks, to the set of processors such that the overall completion time of the application is minimized. The matching and scheduling problem is a very broad problem, taking different forms depending upon the processor architecture, the network architecture, and the task structure. It is also very costly to compute an exact solution to the matching and scheduling problem, since the problem is NP-hard [32]. Therefore, heuristic methods are used to obtain approximate solutions. This report presents a new matching and scheduling heuristic, the *Levelized Min-Time* (LMT) algorithm [21, 22]. To evaluate the performance of this algorithm, and the performance of heterogeneous CSTEM, a series of simulations were performed. These simulations estimate the performance of heterogeneous CSTEM on a variety of potential clusters of heterogeneous machines.

3.1 Definitions

The following set of definitions will be used in presenting the matching and scheduling algorithm. The set of parallel tasks can be represented by a directed, acyclic graph (DAG) $G = (V, E)$, where the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ represents the set of tasks to be executed, and the set of directed edges E represents communication between tasks, where $e_{ij} = (v_i, v_j) \in E$ indicates communication from task v_i to v_j. The collection of heterogeneous machines used in the computation can be represented by the set $P = \{p_1, p_2, \ldots, p_q\}$.

The computation cost matrix $X_{n \times q}$ represents the execution costs of n tasks on q heterogeneous machines. The value $x_{ij} \in X$ represents the computation cost of task v_i on machine p_j. The communication matrix $C_{n \times n}$ holds the number of bytes sent between the tasks. The value of $c_{ij} \in E$ is equal to the communication volume if $e_{ij} \in E$, otherwise, $c_{ij} = 0$.

A solution to the matching and scheduling problem is defined as $\mu : V \rightarrow P$, matching the tasks onto the heterogeneous machines. Thus, task v_i is mapped onto machine $\mu(v_i)$. The communication cost function $\delta : \mathcal{N} \times P \times P \rightarrow \mathcal{N}$ defines the communication costs of a given matching, where $\mathcal{N}$ is the set of natural numbers. The value $\delta(c_{ij}, \mu(v_i), \mu(v_j))$ represents the cost of sending c_{ij} bytes from task v_i on processor $\mu(v_i)$ to task v_j on processor $\mu(v_j)$.

A path W through the DAG $G = (V, E)$ is defined as a sequence of nodes such that, for all adjacent pairs nodes v_i and v_j in the sequence (v_i is ordered before v_j), $e_{ij} \in E$. The cost ϕ of a solution to the matching and scheduling problem, for a given matching, is defined as the path W through the graph that maximizes the sum of the communication costs and computation costs along that path. This can be represented

by the expression:

$$\phi = \max_{W}\left(\sum_{v_i \in W} x_{i\mu(v_i)} + \sum_{v_i, v_j \in W} \delta(c_{ij}, \mu(v_i), \mu(v_j))\right) \quad (1)$$

3.2 Previous Work

There are a wide variety of different approaches that have been taken solve the matching and scheduling problem. The methods that have been used include iterative methods [38], global optimization methods [37, 28], greedy selection methods [10, 20], hierarchical methods [5, 8], and combination methods [6]. Many of these methods do not explicitly consider the precedence relations that exist between tasks, and instead concentrate on matching tasks onto the processors. These methods, in most circumstances, are not applicable to the type of task system defined above.

Therefore, methods which directly consider precedence relations will be emphasized. These methods can be broken down into two categories: those for homogeneous processor systems [34, 20, 40, 9, 1] and those for heterogeneous processor systems [11, 26, 36]. The remainder of this section will examine some of the relevant homogeneous and heterogeneous methods.

For homogeneous task systems, Sarkar and Hennessy [34] present a two-stage technique known as internalization, which first clusters the tasks into an arbitrary number of groups, and assigns these groups to the physical processors. Hwang et al. [20] present a heuristic called earliest task first (ETF), which uses a greedy selection to schedule tasks in homogeneous processor systems. Yang and Gerasoulis [40] present the DSC algorithm, which, on an unbounded number of processors, produces better results than either of the methods presented in [34] or [20]. Colin and Chrétienne [9] present a polynomial algorithm for optimally scheduling tasks on a homogeneous array of processors, provided task duplication is allowed. This method uses a critical path based algorithm. Atallah et al. [1] examine a method for balancing a background computation across a cluster of distributed, homogeneous workstations.

For heterogeneous processor systems, Kim and Browne [26] present a technique called linear clustering, which clusters tasks into chains of tasks, and maps the clusters onto the physical machines. El-Rewini and Lewis [11] present an algorithm known as the MH algorithm. This algorithm prioritizes the tasks based upon an estimate of the starting time, and assigns the tasks based upon those priorities. Both of these heterogeneous methods have limited application to the problem formulated here, since they assume that the individual processors perform uniformly for all code types (i.e. the performance of a task on each heterogeneous processor varies only by a scale factor). This assumption leads to sub-optimal results when applied to a heterogeneous system composed of a diverse range of machine architectures. Sih and Lee [36] present a technique for matching and scheduling in heterogeneous processor systems called

Dynamic Level Scheduling, which assign a series of dynamically changing priorities to the tasks being scheduled. This method is very similar to the technique used by El-Rewini, although it uses a more robust assumption about the nature of the heterogeneous processors, avoiding the problems associated with the MH algorithm.

3.3 Matching and Scheduling with CSTEM

Since many matching and scheduling algorithms are optimized for specific types of problems, when searching for an algorithm, there are several characteristics that need to be matched to the problem. Obviously, with heterogeneous CSTEM, any algorithm must either support heterogeneity or be capable of being extended to support heterogeneity. Other important details include finding an algorithm suitable for both the type of network and the type of machine used. The heterogeneous environment used in this project consists of a variety of general purpose workstations and supercomputers, connected by either a shared medium, like ethernet, or a completely connected packet switched medium, like an ATM switch. In either case, the communication cost between nodes is, under most circumstances, independent of the physical locations of the sending and receiving processors. This type of network is also known as a uniform network.

An algorithm must also be suitable for the type of tasks to be allocated. The amount and grain size of the parallelism, combined with the number of precedence relations between individual tasks, plays a key role in the performance of any algorithm. When parallelizing an existing application, the tasks tend to have a large number of precedence constraints and a relatively low degree of parallelism between them. Precedence is the single most limiting factor to the overall performance of the algorithm; therefore, an algorithm that handles precedence well is essential.

3.4 The Levelized Min-Time Algorithm

The combination of precedence constraints and variable execution times complicates the assignment process. To simplify the problem, a two phase approach will be used. The first phase reduces the precedence constrained matching and scheduling problem into a series of non-precedence constrained sub-problems. The technique that will be used to accomplish this is known as level sorting [31, 7]. Once the problem has been divided using this technique, a much simpler algorithm can be used to solve the individual sub-problems. This algorithm is called the *Min-Time* algorithm. Together, these two stages form the *Levelized Min-Time* (LMT) algorithm.

Considering each subproblem to be completely independent does cause some inaccuracies to be introduced into the solution. Therefore, in order to improve the quality of the solution, some techniques will be given that will include some information from the other subproblems.

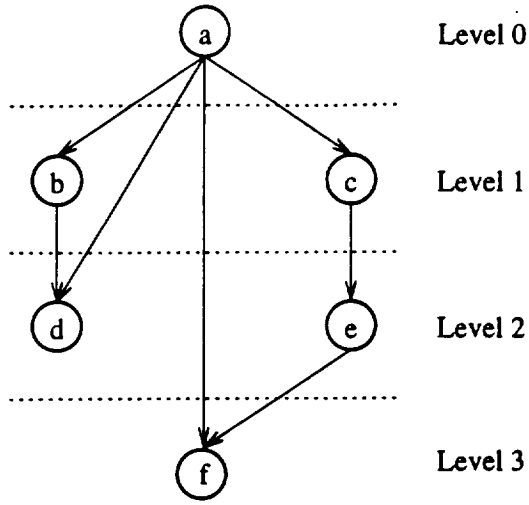


Figure 3.1: Level Sorting Example.

3.4.1 Level Sorting

The method used for the first phase is a technique for ordering the nodes based upon their precedence constraints, called level sorting. Level sorting has applications in several different areas, including logic simulation, fault simulation, and scheduling [31, 7].

The exact definition of the level sorting process can be given recursively: Given a graph $G = (V, E)$, level 0 contains all vertices v_j such that there is no vertex v_i with $e_{ij} \in E$. (i.e. v_j does not have any incident edges). Level k consists of all vertices v_j such that, for all edges $e_{ij} \in E$, every vertex v_i is in a level less than k, and at least one vertex is in level $k-1$. Figure 3.1 shows a sample DAG that has been level sorted.

The level sorting technique clusters nodes that are able to execute in parallel. By clustering tasks in this fashion, the tasks within each level have no precedence constraints between them. The second stage of the LMT algorithm will assign tasks level by level, using an assignment heuristic which does not use precedence information.

3.4.2 Min-Time

The second stage of the assignment process uses a heuristic called the *Min-Time* algorithm. The *Min-Time* algorithm is a greedy method that attempts to assign each task to the "best" processor—the processor on which the task runs the fastest.

The algorithm operates according to the following steps. First, the average execution time of each task, across all available machines, is calculated. Second, if the number of tasks is greater than the number of available processors, the number of tasks is reduced by merging the smallest tasks (based on the average time) until the

number of tasks is equal to the number of processors. Third, the tasks are sorted in reverse order (largest first) by the average execution time. Finally, each task is assigned, in sorted order, to the processor on which it executes the fastest, with at most one task per processor. The sorting process increases the likelihood of large tasks being assigned to the fastest processors, while less demanding tasks are assigned to slower processors.

3.4.3 Final Considerations

Above, the assumption was made that the matching and scheduling problem could be decomposed into a number of independent subproblems. In reality, these problems are not completely independent. The interactions between tasks in different levels can affect the overall cost of a matching. This section will present some features that improve the quality of the solutions.

First, when making an assignment, there is a possibility that the *Min-Time* algorithm might have to choose between two or more identical machines. An effective way to resolve such a choice is to assign the task to the processor from which it receives most of its data, since the cost of sending a byte to another machine is significantly higher than the cost of communicating that byte locally. Therefore, by including the additional information regarding communication between the separate levels, the overall solution can be improved. The simplest way to add this information is to include the cost of communicating with tasks in previous levels into the overall execution cost of each task, increasing the likelihood that tasks which share a large amount of data are assigned to the same processor.

Another drawback with the original assumption results from the *Min-Time* algorithm failing to look at tasks in subsequent levels. The algorithm assumes that in order for a task on level $i+1$ to begin, every task on level i must be complete. In reality, the results produced by task i may not be used for several successive levels, giving that task more time to execute without creating a bottleneck. For example, consider the graph shown in Figure 3.2. If task C is large when compared to task B, under the LO algorithm task C would automatically get priority for assignment to the fastest processors. However, the results produced by task C are first used by task G in level 5. Therefore, Task C will not produce a performance bottleneck unless its execution time is greater than the sum of the execution times of tasks B, D, E, and F. There is a significant problem that occurs when trying to incorporate this information into an algorithm: the execution times of tasks D, E, and F are unknown when tasks B and C are being assigned. Furthermore, in a more realistic example, there would be more than one task in levels 2, 3, and 4, making the the time when task G will begin even more difficult to compute. To solve this problem, a method is needed to establish a reasonable estimate of the execution time of each subsequent level. The method used in this algorithm to estimate this time, is to use the average

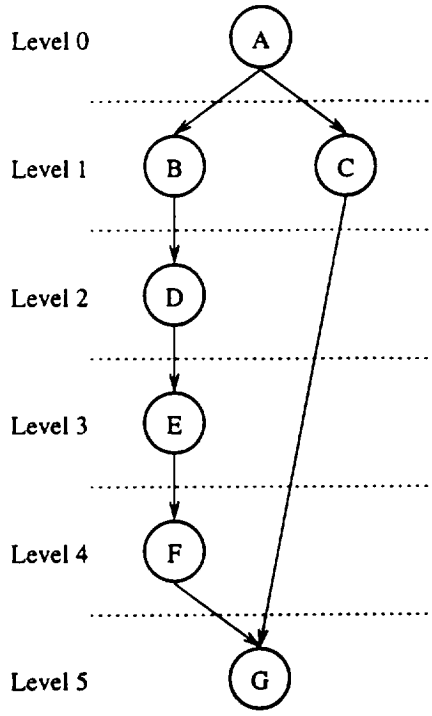


Figure 3.2: A Task Graph Showing the Drawback of Not Examining Subsequent Levels.

time of all of the tasks in the level as an estimate of the execution time of that level. Therefore, if the results from a task in level 1 are not used until level 5, the sum of the average times for the levels 2 through 4 can be used as an estimate of the extra time that the task has to complete execution.

By including the above factors, the formal definition of this algorithm is:

Procedure: LMT
begin
Level sort tasks.
For each level, in order, do:
begin
Assign tasks in level i using *Min-Time*.
end
end

Procedure: Min-Time
begin
For each task v_i do:

```
    begin
        Let avg_i = average value of x_ij for all possible j.
        Adjust average values based upon when results will be
          needed.
    end
    Sort groups in reverse order by avg_i.
    For each task v_i in sorted order do:
    begin
        Find j such that processor p_j does not have a task assigned
          to it and x_ij + Σ_k δ(c_ki, μ(v_k), p_j) is minimal.
        Assign task v_i to processor p_j.
    end
end
```

3.5 Experimental Results

The intended execution environment for heterogeneous CSTEM is the Advanced Computational Concepts Laboratory (ACCL) at the NASA Lewis Research Center. This laboratory consists of a variety of high performance workstations and parallel machines connected by several different high performance networks. To evaluate potential heterogeneous clusters within this environment while limiting the overall programming effort, a series of simulations were performed. A custom event-based simulator, using timing information derived from actual measurements (shown in Figure 2.7) on a Sun Microsystems Sparc 10 workstation, was used to generate these results. The network timings were measured from standard PVM over a conventional ethernet based network. In addition to execution time and network speeds, the simulator also considers the effects of multiprogramming in its computations. The high communication overhead of PVM over a conventional network greatly affects the performance of CSTEM. The setup time for communication using PVM is exceptionally high, clearly creating a performance bottleneck. This problem can be reduced by using a more advanced network with an optimized version of PVM. Several of these networking technologies are available at ACCL, including ATM and other high speed switching technologies.

The results presented in this section are not intended to perform an accurate evaluation of the LMT algorithm, but to explore the potential performance of heterogeneous CSTEM. Given the amount of task parallelism present in the task graph, heterogeneous CSTEM can only effectively utilize about three machines in parallel. Therefore, Figure 3.3 shows the overall speedup obtained by applying the LMT algorithm to heterogeneous CSTEM, using various three machine heterogeneous clusters. The machines in these clusters could potentially be any type of machine, including workstations, parallel machines, or vector supercomputers. One trace shows the es-

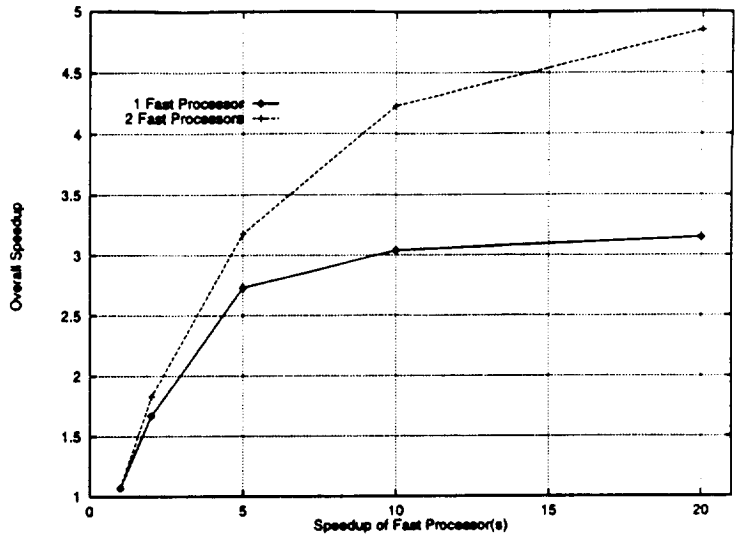


Figure 3.3: Speedup of Heterogeneous CSTEM with Different Processor Speeds.

timated speedup when one of the three machines is either 2, 5, 10, or 20 times faster than the baseline machine. The second trace shows the speedup when two of the machines are either 2, 5, 10, or 20 times faster than the reference machine. The network timings are that of a conventional ethernet based network.

As predicted, these results show that heterogeneous CSTEM is very communication bound, clearly indicating the need for a more advanced network architecture. It is also clear from the structure of the task graph shown in Figure 2.7 that there is a limited amount of parallelism present in the task graph. An effective solution to this problem is to exploit data parallelism within individual tasks.

In the above simulation, it is assumed that any task can execute on any processor. This assumption may not be valid in a real heterogeneous environment, since it may not be worth the programming effort to port non-critical tasks to every available architecture. Therefore, to demonstrate a more realistic example, Figure 3.4 shows the overall speedup obtained from increasing the speed of the five most computationally intensive tasks (M, F, E, K, and N in Figure 2.7) by a factor of 20, as well as the effects of increasing the speed of the network by a factor of 2, 4, and 8. These results show that, given adequate network resources, heterogeneous computing has the potential to provide a significant speedup, while limiting the programming effort to the most computation intensive tasks.

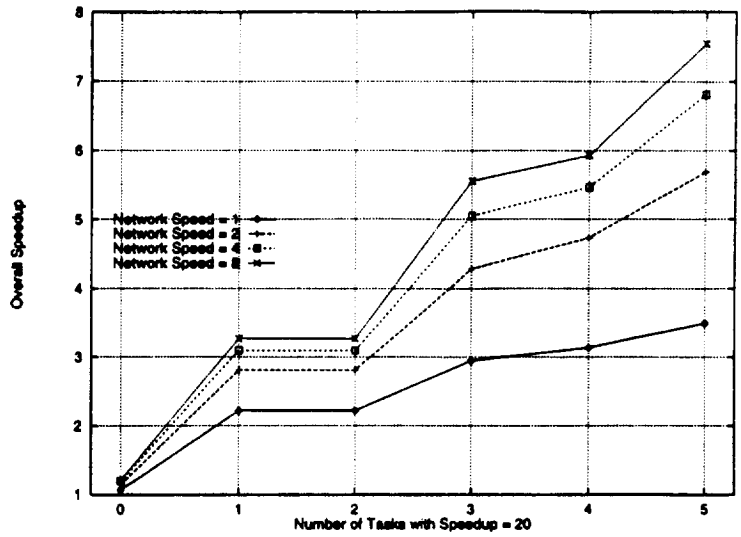


Figure 3.4: Speedups for Different Processor and Network Speeds

3.6 Summary

An effective matching and scheduling algorithm is an essential element of heterogeneous computing, especially one which can take advantage of the available architectural features of a given heterogeneous processing system. In this section, we have presented the the LMT algorithm, an effective matching and scheduling algorithm for heterogeneous computing. We have shown, through simulations, that that heterogeneous computing is an effective means of increasing the performance of large, scientific applications. However, since a matching and scheduling algorithm requires and accurate set of execution time estimates to make effective scheduling decisions, we will, in the next section, present a method for obtaining this set of estimates.

4. Execution Time Estimation

In order for a matching and scheduling algorithm to make a scheduling decision, an accurate set of estimates of the execution time of the task on each potential machine is needed. It is well known that the execution time of a computer algorithm is a function of the size and properties of the input data. The order of this function is known as the *computational complexity*. In the homogeneous case, it can be assumed that each particular task performs identically on each target machine. Therefore, a single estimate of the execution time of each task is required, and is fairly easy to obtain. This, however, is not true for heterogeneous distributed computing, since an execution time estimate is required for each task-machine pair, and there are many factors unique to heterogeneous systems which can affect the execution time, including processor architecture, processor speed, memory size and speed, and machine loading. Furthermore, the heterogeneous environment can be dynamic (unlike many dedicated parallel machines, where the user is granted exclusive use of a portion of the machine) and some of the factors can only be determined at run time. Therefore, it may be be advantageous to have a scheme which could estimate the execution time just before the task is to execute. In this way, the run-time factors could be considered when making the estimate. However, the primary disadvantage of run-time estimation is the speed in which the estimate must be computed. At best, all of the execution time estimates, as well as the matching and scheduling decision, need to be determined in a very small time window, in order to prevent the scheduling overhead from affecting the overall program performance. So, any run-time scheduling technique will have to be computationally efficient.

To meet these requirements, we propose an execution time estimation algorithm which statistically estimates the execution time using past observations [23]. This approach offers a number of advantages. First, a statistical method can compensate for many different factors, without requiring a distinct model for each of the different machine architectures. Second, statistical estimates will improve with time, as the number of previous observations increases. Finally, statistical schemes can be made to be computationally efficient, making them practical for use at run time. One potential criticism of statistical schemes is the need to have a large number of past observations to obtain accurate estimates. This issue will be addressed in Section 4.4.

The method we will present is based upon the statistical technique known as nonparametric regression. Nonparametric regression has the advantage of being able to estimate the execution time, as a function of several parameters, without any knowledge of the function itself. Since we make no assumptions on the functional form, this prediction scheme does not require any knowledge of either the task or the target architecture, making it applicable in a very general sense. This estimation scheme operates in the following manner. A set of previous observations of the execution

time of a task on each potential machine or class of machines is maintained by the algorithm. Using this set of observations, the execution time of a task on each potential machine can be estimated, and the matching and scheduling algorithm can use these estimates to make a scheduling decision. After the task execution is complete, the actual time taken to execute is added to the set of observations, to be used to improve future predictions. By storing past observations, the estimation algorithm is able to improve its estimates over time.

Before we examine the details of this proposed method, we will examine some of the previous work relevant to this paper in the next section. In Section 4.2, the statistical methods used to solve the execution time estimation problem will be discussed. In Section 4.3, the specific details of the method used to solve the execution time estimation problem will be presented. Finally, Sections 4.4 and 4.5 will discuss the experimental results and conclusions drawn from these results, respectively.

4.1 Previous Work

Most of the previous work in execution time estimation for heterogeneous distributed computing centers on a theoretical framework known as analytical benchmarking/code profiling. In *analytical benchmarking*, the source code of a task is analyzed to obtain some set of parameters summarizing the behavior of the task. This technique is used in conjunction with *code profiling*, where the behavior of each machine is summarized in another set of parameters. Once these steps are complete, the information from the tasks and machines can be combined to create an execution time estimate. This framework, first proposed by Freund [14], has appeared often in the literature [39, 24]. Currently, this method is still only a framework, and is far from being a real implementation.

However, other work has been performed in the area of execution time estimation, particularly in reference to estimating execution time directly from the source code. Often, these methods are either targeted for a specific subset of architectures, or are meant to estimate the execution time of source code without reference to a particular architecture. This category includes the work by Li et al [27], who present a method for obtaining execution time estimates for SIMD/SPMD mixed-mode heterogeneous architectures, and Reistad and Gifford [33], who present a method for determining expressions for execution costs, for use with optimizing compilers and for automatic parallelization.

Several authors have applied statistical and probabilistic techniques to distributed computing problems. The SmartNET heterogeneous scheduling tool offers a statistical execution time estimation technique, but no details of its implementation have yet been published [25]. Hou and Shin [19, 35] present techniques, based on Bayesian decision theory, for load sharing in both homogeneous and heterogeneous distributed real-time systems. These methods load balance such that the probability of a real-time task meeting its deadline is maximized. Some of the methods Hou and Shin

present have useful applications in heterogeneous distributed computing, particularly the use of Bayesian decision theory for estimation of the state of individual processors. Although the techniques are not directly applicable to the execution time estimation problem, it would be desirable to use some of Hou and Shin's techniques in conjunction with the method presented here.

4.2 Nonparametric Regression

For the execution time estimation problem defined in this paper, the execution time of a task is considered to be a function $m(x)$ of a parameter x. For example, x could be the problem size. While the estimation algorithm does not know any details about the functional form of $m(x)$, it does have a set of n previous observations of the execution time $\{(y_i, x_i)\}_{i=1}^{n}$, where y_i is the observed execution time for the parameter value x_i. These observations are assumed to contain some amount of random error ϵ_i, such that

$$y_i = m(x_i) + \epsilon_i \tag{1}$$

The goal of the execution time estimation problem is, for some given value of the parameter x, to obtain an estimate $\hat{m}(x)$ of the execution time, using the set of previous observations. In statistics, this problem is called a regression problem.

In a regression problem, there is a system y that is a function of some parameter x, following the form of equation 1. The function $m(x)$ is the *regression function*, and ϵ is zero-mean, random error. Both x and $m(x)$ are deterministic values, while the random error ϵ is stochastic, making y stochastic. In order to simplify the presentation, we will consider x to be a scalar value, although all of the methods presented here can be extended to support a vector of parameters. Estimation with a vector of parameters will be discussed in Section 4.5. The statistical techniques shown in this section and the next, as well as the notation, are derived from the methods collected in the books by Härdle [18] and Eubank [13].

There are a variety of different techniques to solve this problem, that can be divided into two classes: *parametric* techniques and *nonparametric* techniques. In the parametric case, it is assumed that the functional form of $m(x)$ is known. For example, $m(x)$ may be a fourth order polynomial, and the regression problem would be to determine the coefficients of that polynomial. A popular parametric technique for solving this type of problem is the least squares method. It is important to emphasize that the functional form must be correct in order to obtain a meaningful result from a parametric regression technique. Otherwise, inaccurate results may be produced. Since, for the execution time estimation problem defined above, it is difficult to make any assumptions on the functional form of $m(x)$ without specific knowledge of the task and the machine in question, parametric techniques are not well suited to this problem. Nonparametric techniques are a better choice.

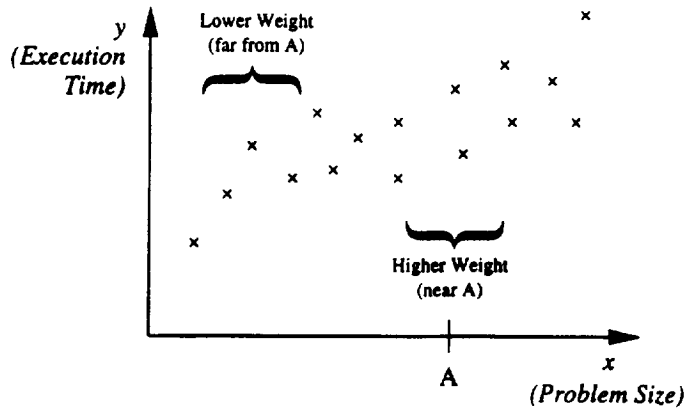


Figure 4.1: Assigning Weights to Observations.

Nonparametric regression techniques (also called nonparametric estimators) make no assumption on the functional form of $m(x)$, and therefore can be considered to be *data driven*, since the estimate $\hat{m}(x)$ only depends upon the set of previous observations. Nonparametric techniques are also called smoothing techniques, since they act to smooth out variations in the observed data caused by the random error ϵ.

All nonparametric regression techniques either follow or can be modeled by

$$\hat{m}(x) = \frac{1}{n}\sum_{i=1}^{n} W_i(x) y_i \tag{2}$$

where $W_i(x)$ is a weighting sequence [18]. From this function, we see that $\hat{m}(x)$, for any given value of x, is a weighted average of the y values of the previous observations. The weight function $W_i(x)$ is a function of x, since it will assign higher weights to observations close to the parameter x, and lower weights to observations farther away from x. This is illustrated in Figure 4.1. In practice, many nonparametric regression techniques only include points within some neighborhood of the parameter x in the average, assigning a weight of "zero" to observations outside of this neighborhood. This makes the estimate $\hat{m}(x)$ a "local" average of the observations near the value of the parameter x. To relate this to the execution time estimation problem, the estimate $\hat{m}(A)$ of the execution time for $x = A$, will be a weighted average of the observations y_i which have parameter values x_i close to the value A.

There are a number of factors to consider when choosing an appropriate nonparametric regression technique. It is important to notice that care must be taken when forming the local average to compute $\hat{m}(x)$. If too many observations are included in the average, the result will be overly biased, making the resulting curve too smooth. On the other hand, if too few observations are averaged, the result will be subject to the variations of the individual samples, making the curve too "noisy."

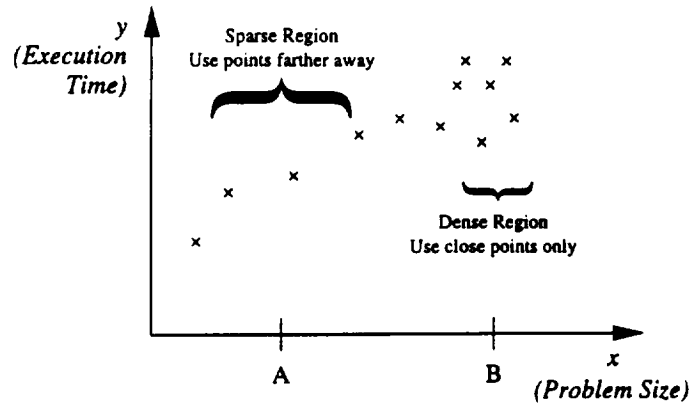


Figure 4.2: Compensating for Observation Density.

Furthermore, as the number of observations tends toward infinity, it is desirable that the estimate $\hat{m}(x)$ approach the true value $m(x)$. This problem is known as the *bias-variance tradeoff*, and is present in all nonparametric regression techniques.

Another factor to consider is the density of the observations. If the observations are not uniformly distributed on the x-axis, the technique needs to be able to compensate for sparse and dense regions of observations. In the dense regions, the average should only include points very close to the parameter value x, while in the sparse regions, points further away from x should be included in the average. This is illustrated in Figure 4.2. One way to accomplish this is to use a fixed number of points in the average. In the next section, we will present an estimation method which uses this technique to compensate for variations in the density of the observations.

4.3 Proposed Estimation Method

The regression technique used for the execution time estimation problem in this paper is based upon a technique known as *k-Nearest Neighbor (k-NN) smoothing*. In *k-NN* smoothing, the estimate $\hat{m}(x)$ for the parameter value A is constructed from the k observations with x values closest to the parameter A. With regard to the execution time estimation problem, there are two primary advantages of *k-NN* smoothing. First, since the estimate is always constructed from an average of k points, the method can easily adapt to sparse or dense regions in the observations. Second, the method can be implemented in a computationally efficient manner.

With uniform weights, the *k-NN* estimator can be formally defined using the weight function

$$W_i(x) = \begin{cases} \frac{n}{k}, & \text{if } i \in J_x \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $J_x = \{i : x_i$ is one of the k nearest neighbors of $x\}$. This weight function is used in equation 2, forming the basic *k-NN* estimator. However, to improve the performance of the method, weights can be assigned to the k observations in the average, based upon the distance of each observation from the parameter value x, with the points closer to the parameter x getting higher weights. A weighting function, (also called a kernel function) with certain optimality properties [12, 13] is the Epanechnikov Kernel $K(u)$, where

$$K(u) = \frac{3}{4}(1 - u^2) \tag{4}$$

and $|u| < 1$. To incorporate this kernel function into the *k-NN* estimator, we need to ensure that it is properly scaled and normalized. To accomplish this, let

$$W_i(x) = \begin{cases} \frac{K_R(x-x_i)}{\hat{f}_R(x)}, & \text{if } i \in J_x \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $K_R(u)$ is the scaled Epanechnikov kernel

$$K_R(u) = \frac{1}{R}K(\frac{u}{R}). \tag{6}$$

The Epanechnikov kernel is scaled by the factor R, which, for the k points in J_x, is defined to be

$$R = \max_{J_x}(x - x_i). \tag{7}$$

Finally, the factor $\hat{f}_R(x)$ in equation 5 is a normalizing factor, defined as

$$\hat{f}_R(x) = \frac{1}{k}\sum_{J_x} K_R(x - x_i). \tag{8}$$

This weighted estimator will be used to solve the problem presented in this paper. However, in order to further improve the performance of this estimator, we will define some additional modifications.

4.3.1 Boundary Effects

A factor that needs to be accounted for is the behavior of the regression technique at the boundaries of the set of previous observations (i.e. no observations lie beyond the boundary). As x approaches a boundary, the local average becomes biased, since more observation points will be on one side of point x than the other. This is illustrated in Figure 4.3, where the estimated function $\hat{m}(x)$ will tend to "fall away" near the boundary [13, 18]. Therefore, a nonparametric regression technique should be able to compensate for this effect.

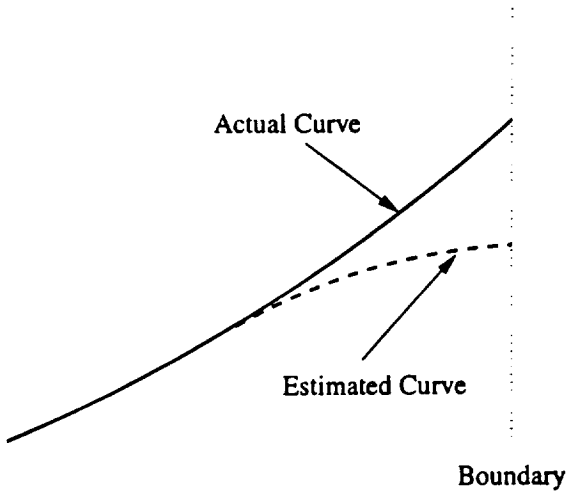


Figure 4.3: Effects of Estimates at the Boundary.

One computationally efficient method of compensating for the boundary effects is to ensure that the interval from which the points in the local average are selected is "evenly spaced," where x always lies at the center of the interval. For example, if the observations are bounded to the interval $[a, b]$, and the value of x is near b, we can restrict the points in the local average to be from the interval $[x - (b - x), b]$. Similarly, if the value of x is near a, we can restrict the points in the local average to be from the interval $[a, x + (x - a)]$. To accomplish this, we can formally redefine the set J_x to be

$$\begin{aligned} J_x &= \{i : x_i \text{ is one of the } k \text{ nearest neighbors of } x\} \cap \\ &\quad \{i : x_i \in [a, x + (x - a)]\} \cap \\ &\quad \{i : x_i \in [x - (b - x), b]\}. \end{aligned} \tag{9}$$

By defining the set in this fashion, the local average will be more likely to have an equal number of points above and below the parameter value x. The disadvantage of this technique is that estimates close to the boundary will have a higher variance, because fewer points are used to compute the average. However, the higher variance is preferable to the biased estimates, since, as the number of observations grows, the variance will decrease.

4.3.2 Robustness

Another desirable factor to consider is how the technique behaves when erroneous data points are included in the data set. These points, called outliers, do not conform

to the model described in equation 1. These outliers may end up in the data set due to erroneous readings, an overloaded machine, or due to other poorly modeled effects. An estimator with the ability to disregard these points is called a *robust estimator*. One suitable technique to accomplish this is called L-Smoothing [18], where a set percentage of the observations with the largest and smallest values of y_i are eliminated from the local average. L-Smoothing can be implemented by sorting the observations $\{(x_i, y_i)\} \in J_x$ by y_i, then computing

$$\hat{m}(x) = \frac{1}{k - 2\lceil \alpha k \rceil} \sum_{i=\lceil \alpha k \rceil}^{k-\lceil \alpha k \rceil} W_i(x) y_i \tag{10}$$

The value of α, where $0 < \alpha < 0.5$, controls the percentage of observations excluded from the average.

4.3.3 Asymptotic Behavior

As described in Section 4.2, there is a tradeoff between the bias and the variance of the estimated curve. If too many points are used in the *k-NN* average, the bias $E\{\hat{m}(x) - m(x)\}$ will be too large, while using too few points in the average will cause the variance $E\{\hat{m}^2(x)\}$ will be too large. Therefore, the value of k must be chosen with reasonable care. Also, the value of k depends upon the number of sample observations n, and, therefore, needs to be adjusted accordingly. It has been shown that, by increasing k in proportion to $n^{\frac{4}{5}}$, the *k-NN* technique will maintain a constant tradeoff between the variance and the bias [18, 29].

4.3.4 Computational Complexity

The methods presented in this section can be implemented in a computationally efficient manner. Excluding the robust, L-smoothing technique, this estimator can be implemented with an algorithm which is linear in the parameter k. Since k is set to be proportional to $n^{\frac{4}{5}}$, this makes the overall algorithm $O(n^{\frac{4}{5}})$ with respect to the number of observations n. The L-smoothing estimator is not quite as efficient, since the observations $\{(y_i, x_i)\}_{i=1}^{n}$ must be sorted by the value of y_i, which is an $O(k \log k)$ operation, making the overall complexity $O(n^{\frac{4}{5}} \log n)$.

4.4 Results

A number of simulations were performed to evaluate the effectiveness of the proposed method. In these simulations, the execution time is assumed to follow the model described in equation 1. The simulations begin by choosing a function $m(x)$, and computing an initial set of 10 previous observations. Then, in each step, a random value

of parameter x is generated. Using the method presented above, an estimate $\hat{m}(x)$ of the execution time is made. To simulate the execution of the task, the actual "execution time" is computed to be the value of $m(x)$ plus some zero mean random error, as shown in equation 1. Now, given the predicted time and the "actual" time, the prediction error can be computed. Finally, the "actual" execution time is added into the set of observations, and the simulation process repeats. In this fashion, we can observe the behavior of the error as the number of observations (n) increases.

To illustrate the importance of each of the different components described in Section 4.3, we compare the behavior of four versions of the *k-NN* algorithm: using uniform weights only, using uniform weights and boundary compensation, using nonuniform weights and boundary compensation, and using the complete, robust algorithm. The mean absolute error measure is used to evaluate each of the four estimation schemes. This was done because the absolute error is easier to relate to actual measurements, as opposed to the mean squared error.

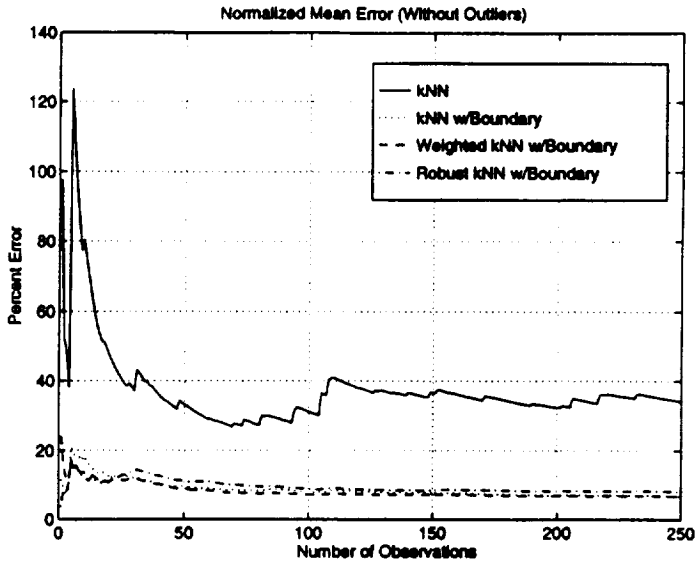
In Figure 4.4, the algorithm is applied to data which does not contain outliers. Figure 4.4(a) shows the normalized mean error, as a function of the number of observations, while Figure 4.4(b) shows the actual curve $m(x)$ and the observations. In this case, the best performing algorithm is the weighted *k-NN* algorithm. The robust algorithm has slightly, but not significantly worse performance, due to the fact that the robust modifications described in Section 4.3 discard some of the points from the average. For this simulation, the mean prediction error over 50 observations falls below 10% for all algorithms except for the simple, uniformly weighted *k-NN* algorithm.

In Figure 4.5, the algorithm is applied to data which contains outliers. In this case, 10% of the observations do not conform to the model described by equation 1. This can be seen clearly in Figure 4.5(b). As in the previous case, Figure 4.5(a) shows the normalized mean error, as a function of the number of observations. In this case, the value of having a robust algorithm is clear, where the full algorithm clearly outperforms all of the others. It is important to observe that this estimation technique produces good results with a small number of observations. With as few as 10 observations, the mean prediction error is less than 20% of the "true value" of $m(x)$. Obtaining ten observations for each task/machine pair is quite reasonable, since these measurements can be made during the testing and debugging of the application. This, coupled with the "learning" capability of this algorithm, make it an effective prediction scheme.

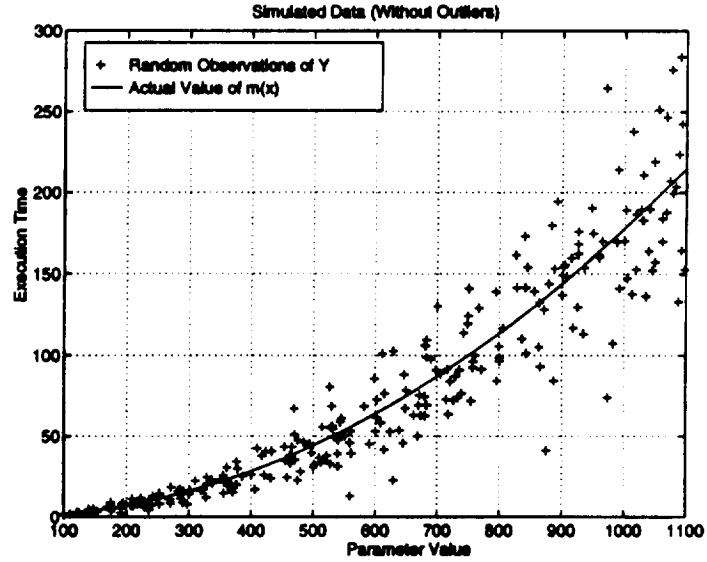
4.5 Further Work and Conclusions

4.5.1 Multidimensional Parameters

Thus far, this paper has only considered execution time estimation as a function of a scalar parameter x. However, it is highly desirable to be able to compute an

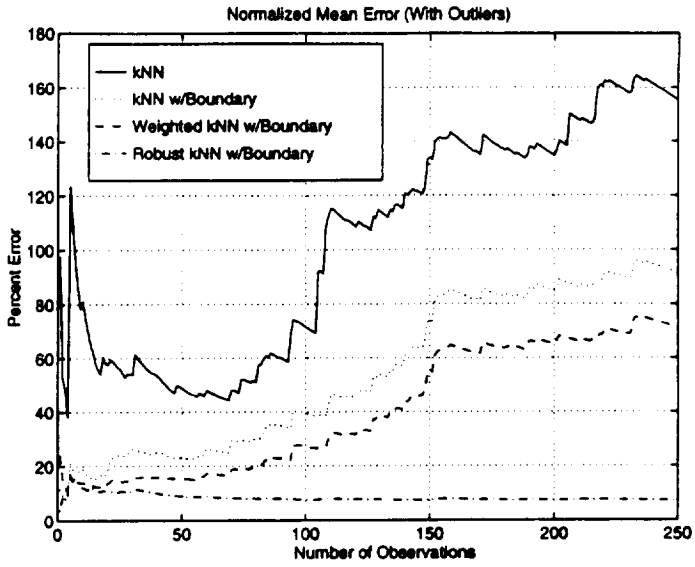


(a) Normalized Mean Error For Different Numbers of Observations

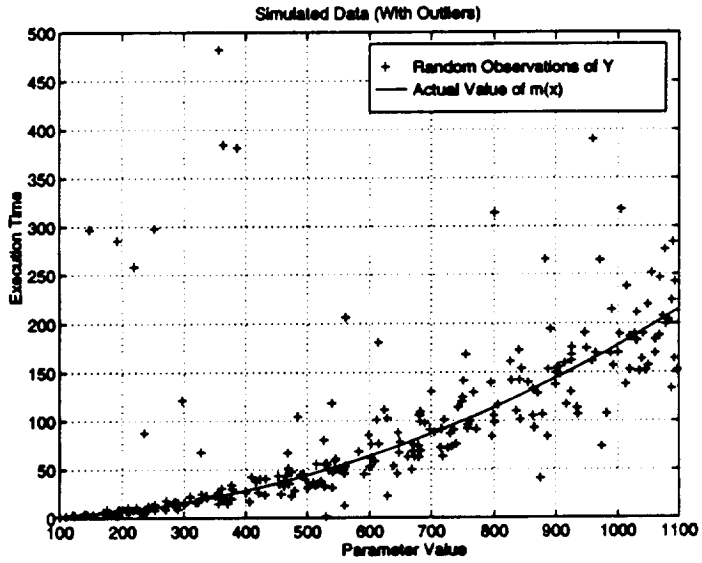


(b) Simulated Execution Time Function and Random Observations

Figure 4.4: Estimator Performance for Data Without Outliers.



(a) Normalized Mean Error for Different Numbers of Observations



(b) Simulated Execution Time Function and Random Observations

Figure 4.5: Estimator Performance for Data With Outliers.

execution time estimate using a vector of parameters $X = [x^1 x^2 \cdots x^m]$. In principle, the task of modifying the methods presented in Section 4.3 to support a vector of parameters is straightforward. Given an m dimensional vector $X = [x^1 x^2 \cdots x^m]$, the execution time estimate is computed from the k observations closest to the vector X in Euclidean distance. All of the equations presented above can be easily redefined in terms of this vector X.

While the multidimensional case is theoretically simple, there is significant impact upon the computational efficiency of the estimation method. This impact is primarily due to the complexity of finding the k nearest neighbors of X, which is a point in m-dimensional space. In the scalar case, the observations can be stored in an array sorted by their x value, making the process of finding the k nearest neighbors a simple task. This storage scheme cannot be easily be modified to suit a multidimensional implementation, greatly increasing the computational cost of computing the estimate.

To improve the efficiency of the proposed estimation method for multidimensional parameters, we propose a scheme which will obtain an estimate by interpolating between a set of precomputed values of $\hat{m}(X)$. To accomplish this, a number of estimates for evenly spaced values of X would be computed off-line, and then, at run time, an estimate for an arbitrary value of X can be obtained by interpolating between the precomputed values. When the task completes execution, the actual execution time would be stored, and, at a later time, would be added into the existing observations, and the values of $\hat{m}(X)$ would be recalculated. The actual interpolation method could either be a simple linear interpolation, or a more accurate (and costly) higher order approximation. Although experimental results are incomplete, the linear interpolation will probably produce estimates with sufficient accuracy.

The principal advantage to this approach is the reduction of the on-line computation costs. This scheme would even reduce the computation cost of execution time estimate with a scalar parameter. The disadvantages of this scheme are that new observations are not immediately incorporated into future estimates, and that off-line processing of the data is required. The impact of the off-line processing is not significant, since the estimates can be updated with new observations either at off-peak times or by running the task in the background at a low priority. The impact of not immediately incorporating new observations into future estimates is more significant, particularly for applications with few observations. Therefore, more frequent updates may be required for applications which do not have a sufficient number of previous observations.

A minor modification to the interpolation based scheme can improve the estimates for observations which lie beyond the range of the current set of observations. Since the nonparametric estimation scheme presented in this paper computes the execution time estimate from previous observations, estimates beyond the range of the current set of previous observations can be inaccurate. However, similar to the interpolation scheme, estimates beyond the range of current observations can be extrapolated from

the set of precomputed results.

4.5.2 Summary

In this section, we have presented an efficient method for estimating the execution time of a task in order to to facilitate efficient matching and scheduling algorithms in a distributed heterogeneous environment. This method statistically estimates the execution time based upon previous observations, compensating for the parameters upon which the execution time depends. This method has been shown to be computationally efficient, and extensions have been proposed to further improve its efficiency. Experimental results show that the execution time estimates are accurate, even when there are relatively few previous values from which to compute an estimate. These features, combined with the ability for the estimates to improve with time, make this method useful for matching and scheduling in a heterogeneous environment.

5. Conclusions

Applications based upon the finite element method are well known for their demand for computational resources. An effective method for satisfying this demand is heterogeneous parallel computing. In this report, we have presented the results obtained by applying heterogeneous computing to a large finite element application: CSTEM. A difficult problem associated with heterogeneous computing is the matching and scheduling problem—the process of assigning the tasks of a parallel program to the individual processors. A simple assignment heuristic, *Levelized Min-Time* (LMT), has been presented, along with simulated results from applying the LMT algorithm to heterogeneous CSTEM on a variety of different heterogeneous machine clusters.

In order to make effective matching and scheduling decisions, an accurate set of execution time estimates is required. Therefore, we have also presented an efficient, run-time, statistical scheme for estimating the execution time of a task, in order to facilitate matching and scheduling in a distributed heterogeneous computing environment. This scheme is based upon a nonparametric regression technique, where the execution time estimate for a task is computed from past observations. This technique is able to compensate for different parameters upon which the execution time depends, and does not require any knowledge of the architecture of the target machine. It is also able to make accurate predictions when erroneous data is present in the set of observations, and has been experimentally shown to produce estimates with very low error, even with few past values from which to calculate a new estimate.

From the results presented above, heterogeneous computing has been shown to have the potential to significantly increase the performance of existing applications. Furthermore, heterogeneous computing offers a number of advantages over other techniques in its ability to take advantage of different architectural features, and, with suitable programming tools, to limit the overall programming effort.

References

[1] M. J. Atallah, C. Lock, D. C. Marinescu, H. J. Seigel, and T. L. Casavant, "Co-scheduling compute-intensive tasks on a network of workstations: Models and algorithms," in *The 1991 IEEE Inter. Conf. Distributed Computing Systems.*, pp. 344–352, July 1991.

[2] K. Bathe, *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, Inc., 1982.

[3] A. Beguelin, J. Dongarra, G. A. Geist, R. Mancheck, K. Moore, R. Wade, J. Plank, and V. Sunderam, *HeNCE Users' Manual*, Dec. 1992.

[4] A. Beguelin, J. Dongarra, G. A. Geist, R. Mancheck, and V. S. Sunderam, "Graphical development tools for network-based concurrent supercomputing," in *Supercomputing 91*, pp. 435–444, The IEEE Computer Society Press, Nov. 1991.

[5] N. S. Bowen, C. N. Nikolaou, and A. Ghafoor, "On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems," *IEEE Trans. Computers*, vol. 41, pp. 257–273, Mar. 1992.

[6] V. Chaudhary and J. K. Aggarwal, "A generalized scheme for mapping parallel algorithms," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 328–346, Mar. 1993.

[7] C. L. Chen, C. S. G. Lee, and E. S. H. Hou, "Efficient scheduling algorithms for robot inverse dynamic computation on a multiprocessor system," *IEEE Trans. Systems, Man, Cybernetics*, vol. 18, pp. 729–743, Sept. 1988.

[8] S. Chen, M. M. Eshaghian, A. Khokhar, and M. E. Shaaban, "A selection theory and methodology for heterogeneous supercomputing," in *Proc. of the 1993 Workshop on Heterogeneous Processing*, pp. 15–22, The IEEE Computer Society Press, 1993.

[9] J. Y. Colin and P. Chrétienne, "C.P.M. scheduling with small communication delays and task duplication," *Operational Research*, vol. 39, no. 4, pp. 680–684, 1991.

[10] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, vol. 15, pp. 50–56, June 1982.

[11] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel Distributed Computing*, vol. 9, pp. 138–153, 1990.

[12] V. A. Epanechnikov, "Non-parametric estimation of a multivariate probability density," *Theory of Probability and Its Applications*, vol. 14, pp. 153–158, 1969.

[13] R. L. Eubank, *Spline smoothing and nonparametric regression.* M. Dekker, 1988.

[14] R. Freund, "Optimal selection theory for superconcurrency," in *Proceedings of the 1989 Supercomputing Conference*, pp. 13–17, The IEEE Computer Society Press, 1989.

[15] R. F. Freund and H. J. Siegel, "Heterogeneous processing," *IEEE Computer*, vol. 26, pp. 13–17, June 1993.

[16] G.E. Aircraft Engines, *CSTEM Users Manual*, Mar. 1992.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Lab., May 1993.

[18] W. Härdle, *Applied nonparametric regression.* Cambridge University Press, 1990.

[19] C.-J. Hou and K. G. Shin, "Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems," *IEEE Trans. Computers*, vol. 43, pp. 1076–90, Sept. 1994.

[20] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times.," *SIAM J. Computing*, vol. 18, pp. 244–257, Apr. 1989.

[21] M. A. Iverson, "Mapping and scheduling in a distributed, heterogeneous computing environment," Master's thesis, The Ohio State University, Columbus, Ohio, 1994.

[22] M. A. Iverson, F. Özgüner, and G. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," in *Proc. of the 1995 Heterogeneous Computing Workshop*, (Santa Barbara, CA), pp. 93–100, Apr. 1995.

[23] M. A. Iverson, F. Özgüner, and G. Follen, "Run-time statistical estimation of task execution times for heterogeneous distributed computing," in *Proc. of the 1996 High Performance Distributed Computing Conference*, (Syracuse, NY), pp. 263–270, Aug. 1996.

[24] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *IEEE Computer*, vol. 26, pp. 18–27, June 1993.

[25] T. Kidd, D. Hensgen, L. Moore, R. Freund, D. Charley, M. Halderman, and M. Janakiraman, "Studies in the useful predictability of programs in a distributed and homogeneous environment.," *The Smartnet Home Page (http://papaya.nosc.mil:80/SmartNet/)*, 1995.

[26] S. J. Kim and J. C. Browne, "A general approach to mapping parallel computations upon multiprocessor architectures.," in *The 1988 Inter. Conf. on Parallel Processing*, vol. 3, pp. 1–8, 1988.

[27] Y. A. Li, J. K. Antonio, H. J. Seigel, M. Tan, and D. K. Watson, "Estimating the distribution of execution times for SIMD/SPMD mixed-mode programs," in *Proc. of the 1995 Heterogeneous Computing Workshop*, pp. 35–46, The IEEE Computer Society Press, Apr. 1995.

[28] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. Computers*, vol. 37, pp. 1384–1397, Nov. 1988.

[29] Y. P. Mack, "Local properties of k-NN regression estimates," *SIAM J. Alg. Disc. Meth.*, vol. 2, no. 3, pp. 311–323, 1981.

[30] R. McKnight, "Coupled disciplinary analysis for aircraft gas turbine engines," in *The 1992 Interdisciplinary System Simulation and Design Workshop*, pp. 7–21, The Ohio Aerospace Institute, 1992.

[31] R. R. Muntz and E. G. Coffman, "Optimal preemptive scheduling on two-processor systems," *IEEE Trans. Computers*, vol. C-18, pp. 1014–1020, Nov. 1969.

[32] M. J. Quinn, *Parallel Computing: Theory and Practice.* McGraw-Hill Book Company, 1993.

[33] B. Reistad and D. K. Gifford, "Static dependent costs for estimating execution time," in *Proc. of the 1994 ACM Conference on LISP and functional programming*, pp. 65–78, The ACM Press, June 1994.

[34] V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," *ACM SIGPLAN*, vol. 21, pp. 17–26, July 1986.

[35] K. G. Shin and C.-J. Hou, "Design and evaluation of effective load sharing in distributed real-time systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, pp. 704–19, July 1994.

[36] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, pp. 175–187, Feb. 1993.

[37] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Engineering*, vol. SE-3, pp. 85–93, Jan. 1977.

[38] L. Tao, B. Narahari, and Y. C. Zhao, "Heuristics for mapping parallel computations to heterogeneous parallel architectures," in *Proc. of the 1993 Workshop on Heterogeneous Processing*, pp. 36–41, The IEEE Computer Society Press, 1993.

[39] J. Yang, I. Ahmad, and A. Ghafoor, "Estimation of execution times on heterogeneous supercomputer architectures," in *The 1993 Inter. Conf. on Parallel Processing*, vol. 1, pp. 219–226, The CRC Press, Aug. 1993.

[40] T. Yang and A. Gerasoulis, "DSC: Scheduling tasks on an unbounded number of processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, pp. 951–967, Sept. 1994.