

On Some Statistical Methods for Parallel Computation

Edward J. Wegman
Center for Computational Statistics
George Mason University
Fairfax, VA 22030

1 Introduction

The history of computing has been marked by the inevitable march toward higher performance computing devices. As the physical limits of electronic semiconductors reach submicron sizes and current movement is limited to a few hundreds of electrons, the search for increased speeds and performance in computing machines has shifted focus from high performance specialized processors once found in past generations of supercomputers to so-called massively parallel computers built on commodity CPU chips. These commodity chips have evolved much more rapidly than specialized processors because the large scale of the installed base allows economies of scale in their development. Parallel computation is an efficient form of computing, which emphasizes the exploitation of concurrent events. Even the simplest microcomputers to some extent have elements of concurrency such as simultaneous memory fetches, arithmetic processing and input/output. However, generally the term, parallel computation, is reserved for machines or clusters of machines, which have multiple arithmetic/logical processing units and which can carry on simultaneous arithmetic/logical operations. The architectures that, when implemented, lead to parallel machines can be thought of in three generic forms: pipeline processors, array processors, and concurrent multiprocessors.

Conventional supercomputers were an evolution of mainframe computers that focused on the development of very powerful custom arithmetic processors. The CRAY Y-MP supercomputer for example at most had 16 processors. The individual arithmetic processors were so physically small and operated at sufficiently high current levels that heat dissipation was a major problem and the systems required liquid nitrogen or other exotic coolants to allow them to function. Because the number of processors is relatively small, they did not easily lend themselves to applications in which there are high data input requirements. In a general sense, while they were, for their era, extremely powerful number crunchers, the number of megaflops per byte of input data had to be comparatively large to keep their arithmetic processors operating at full potential. Cal-

culations involving the solutions to partial differential equations typically have such a character. Thus applications involving aerodynamic and fluid dynamics, high energy physics calculations, computational chemistry, cryptographic analysis and seismic exploration were the primary beneficiaries of supercomputing. Most statistical calculations involve only a relatively modest number of megaflops per byte of data and so do not demand traditional supercomputing capability. Ironically, the current emphasis on data mining only became practical with improvements in data storage and parallel computation. Indeed, data mining applications such as clustering with large data sets demand the highest capabilities of modern computing. See Wegman [1] for a discussion of computational feasibility.

2 The Forms of Parallel Computers

The three generic forms of parallel computing devices are pipeline processors, array processors, and concurrent multi-computers (either as a single machine with many processors or as a cluster of linked independent machines). To understand a pipeline processor, it is important to understand that the process of executing an instruction consists of four major steps: the instruction fetch, the instruction decoding, the operand fetch, and the execution. In a nonpipeline computer these steps must be completed before the next instruction can be issued. In a pipeline machine, successive instructions are executed in an overlapped fashion. An instruction cycle is the amount of time it takes to execute a single instruction. The instruction cycle can be thought of as being made up of multiple pipeline cycles, a pipeline cycle being equal to the time required to complete the (four) stages of instruction execution. In the nonpipeline computer, a single instruction will require four pipeline cycles, whereas, in a pipeline computer, once the pipeline is filled up, an instruction is executed at each pipeline cycle. Therefore, for this example, the pipeline machine is four times as quick as a nonpipeline machine.

Actually a machine need not be limited to a 4-stage pipeline, but could be extended to a k -stage pipeline depending on the machine designer's desire. Theoretically, a k -stage linear pipeline processor could be at most k times as fast. However, because of memory conflicts, data dependency, branching and interrupts, this theoretical speedup will never in practice be achieved. Nonetheless, it is clear that considerable improvement in efficiency can be achieved if the pipeline is tuned to performing a specific operation repeatedly. That is to say, whenever there is a change in operation, for example from addition to multiplication, the arithmetic pipeline must be drained and reconfigured. Therefore, most pipelines are tuned to specific operations, usually addition and multiplication, and used in situations where these operations are performed repeatedly. The prime example of these are vector processors used in traditional supercomputers in which the same operations are repeatedly applied to components of a vector.

An array processor in contrast is a synchronous parallel computer with mul-

multiple arithmetic/logic units (processing elements) that operate simultaneously in a lockstep fashion. By replicating the processing elements in an appropriately configured and connected geometric array, it is possible to achieve a spatial distribution of processing elements, a spatial parallelism. An appropriate data routing mechanism must be constructed for the processing elements. The processing elements may be either special purpose processing elements or general purpose. Machines in this category often contain very large numbers of processors. In the case of special purpose machines such as the systolic array for multiplying $n \times n$ matrices, approximately n^2 processing elements are required. The general purpose case was illustrated by the massively parallel bit-slice processors, a commercial example was the Connection Machine. The Connection Machine could have been purchased with between 16,000 and 64,000 processing elements. Machines with a large number of simple processors were generally known as fine grain machines. Pipeline processors have become the standard for commodity chips such as those manufactured by Intel, AMD, and Motorola and are no longer the province of supercomputers. General and special purpose fine grain machines have fallen out of favor and are no longer commercially produced.

In contrast with the fine grain machines are coarse grain machines, which, as one would guess, have a more limited number of high performance processors together with appropriate memory. Memory may either be shared or local to the processor. In any case the processors are configured with some connection topology that in simplest form may be a ring network, a lattice, a star network, a tree-structured graph, and, in most complicated form, may be a complete graph. A machine with a ring structure economizes on the total number of interconnects required, but exchanges this for distance between nodes. If communication between nodes is important for a given application, typically a ring-connected architecture will have bad performance since distance between nodes is a surrogate for communication time. At the other extreme is a machine with a complete graph. Here every node is connected to every other node leading to fast internode communications but also cumbersome interconnect wiring and clumsy control mechanisms. Some intermediate configurations include the butterfly, the binary tree and the hypercube configuration. In the mid-1980s to early 1990s, a number of commercial parallel machines were sold. Intel introduced a hypercube-based machine in 1985 known as the iPSC (Intel Personal Super Computer) based on the 80286 chip with ethernet connections between nodes. This was later replaced with the iPSC/2 using 80386 chips and the iPSC/860 and Paragon using 80860 risc chips. The later machines had special purpose NICs (network interface chips) that provided direct routing and improved communication overhead dramatically. More recently, networking individual PCs using ethernet connections (Beowulf clusters) has become popular.

In a multiprocessor environment one would like to achieve linear speedup, i.e. for k processors, one would like a k -fold speedup. However, because of communication overhead and inherent non-parallelizability, this is never achieved. More processors make the communication overhead more intense so that there is a practical reason of diminishing returns for limiting total number of processors.

There are several taxonomies of parallel machines emphasizing different features. One classification involves instruction and data streams. An instruction stream is a sequence of instructions as executed by a machine. A data stream is a sequence of data including input, intermediate or partial results called for by the instruction scheme. A machine may involve a single or multiple instruction stream and a single or multiple data stream. The single instruction single data (SISD) is the mode of most commonly available serial machines. The machine may be pipelined, however, so that there can be an element of parallelism even in a SISD machine. In the single instruction multiple data (SIMD) architecture all processing elements receive the same instruction broadcast from the control unit, but have their own unique data sets flowing from different data streams. Typically an array processor is an SIMD. The multiple instruction single data (MISD) involves multiple processors receiving a distinct instruction set but operating on a common data set. No practical examples of this style of computing architecture exists. The multiple instruction multiple data (MIMD) uses multiple instruction streams and multiple data streams. Inherent in this organization is the implication that the multiple processors have substantial interactions and usually share intermediate results. Most parallel computers have this architecture although they are often used in a SIMD mode.

Another classification which is usually applied to multiprocessor machines is the shared memory versus local memory dichotomy. In a shared memory machine, each of the processors accesses the same memory. This may be done through a time-shared common bus, through a crossbar switch network or by means of a multiport memory. The shared memory is the vehicle not only for storage of intermediate results, but a vehicle for internode communications. In a local memory machine, each processor owns and accesses only its own memory. Hence, internode communication must take place along the communication linkages established between nodes. Such machines are frequently referred to as message-passing machines. Because of the popularity of linking independent PCs in Beowulf-type clusters, message passing has become the dominant style of parallel computing. Gropp and Lusk [2], Gropp et al. [3], and Snit et al. [4] are excellent references for parallel computation using message passing. Dongarra et al. [5] and Foster [6] are useful references for general parallel programming.

3 Stochastic Domain Decomposition

Contemporary statistical computations often focus the analysis of massive data sets with complex algorithms. Consequently, efforts to speed up the calculations are extremely important even with the impressive computational power available today. Parallel computation techniques are an important technology that is used to achieve speedup. Historically in the 1960s and early 1970s, parallelism was also used in the design of both hardware and software to enhance the reliability of systems through redundancy. In such a design, components (either hardware or software) run in parallel performing the same task. The parallel processors each process the same data, with a voting procedure used to determine the

reported outcome of the computation. That is, the outcome with the most votes is the one thought to be correct. The object of the redundancy in this case is fault tolerance. Of course, this type of parallelism leads to no inherent speedup in the computations.

One may use parallelism in achieving speedup by sending different data to the different processors. This can result in substantial speedup, depending on communication overhead and the details of the implementation of parallelism. However, in this mode of operation there is no mechanism for achieving fault detection. For example, the decomposition of an integral and assignment of portions of that integral to processors in a numerical quadrature algorithm is an illustration of this sort of parallelism. It would usually be impossible to know whether one processor returned an incorrect value for its portion of the integral. An interesting question then is whether or not there are situations where I can use parallelism for speedup and still maintain some of the properties of redundancy for my reliability checks. In fact, this is possible in some situations, as will be described below.

An important style of parallel computation is to exploit a cluster of machines in a SIMD mode. The same program is sent to each processor, but different elements of the data are sent to each processor. The art of dividing up the data in such a way that the results of the computation may be reassembled into a useful answer is known as domain decomposition. Xu et al. [7] suggested a form of stochastic domain decomposition. The general idea of domain decomposition is to parse the data so that each processor has as nearly as possible data sets of the same size and which will take as near as possible the same amount of time to compute (load balancing). In the numerical quadrature example mentioned above, if there are k processors and say, for example, it is intended to approximate the integral by $k \times 1000$ rectangular strips, then the domain decomposition strategy might be to assign the computations for the 1st strip, the $(k + 1)$ st strip, the $(2k + 1)$ st strip, \dots to the first processor, the 2nd, the $(k + 2)$ nd, the $(2k + 2)$ nd, \dots to the second processor and so on. Thus every processor would have the area of 1000 strips to compute, which then could be summed to approximate the integral.

In a setting in which data may be assumed to be generated from some probabilistically homogeneous structure, the use of statistical hypothesis tests in place of voting procedures to compare results from different nodes is suggested. Data are assigned to nodes by parsing in an appropriate manner. As a first step, I assume that the data may be parsed into random samples. Since I began with stochastically homogeneous data and parsed it into random samples, the only variation in the output that I should expect to see from the different nodes is stochastic variation. Hence, I can use statistical tests to check the results for deviations from homogeneity. These tests yield a stochastic measure of redundancy for my parallel implementation. I can, thus, use the tests for fault detection in either of the node hardware or software. This form of domain decomposition is stochastic domain decomposition. I will describe several examples of using this methodology.

3.1 General Purpose Stochastic Domain Decomposition

The general purpose stochastic domain decomposition relies on the Central Limit Theorem and so is appropriate for relatively massive datasets. See, for example, Wegman [1]. Consider a random sample X_1, \dots, X_n where n is a large number, for example 10^6 to 10^{12} . Suppose also that I have k nodes. Typically k might be somewhere on the order of 64 to say 1000, but very much smaller than n . The basic idea is to operate the machine in a SIMD fashion so that each node is running the same algorithm, but applying it to stochastically equivalent sets of data. The idea is to spawn m processes on each node where m is a relatively large number. Then $m \cdot k$ is the total number of processes and $n/(m \cdot k)$ is the number of observations per node. For sake of simplicity, let me temporarily assume that $n/(m \cdot k)$ is an integer. If not $\lfloor (n/(m \cdot k)) + 1 \rfloor$ will be an integer, where $\lfloor \cdot \rfloor$ is the greatest integer function, and the algorithm will require a slight modification.

If the computation is a linear functional, e.g. mean, sum of observations, sum of squared observations (hence variance), kernel density estimator, or similar linear computation, then the final computation can be made by summing the components from each process on each node. Hence there is a possibility of substantial speedup based on this stochastic domain decomposition. Further since each process on each node is operating on stochastically equivalent independent data with the same algorithm, I essentially have a sample of size m from each of the k nodes, so that I could form the mean value $\hat{\mu}_i$, $i = 1, \dots, k$ for each of the k nodes. By the CLT, these $\hat{\mu}_i$ will be approximately normally distributed and so I can apply a standard ANOVA to test $H_0 : \mu_1 = \dots = \mu_k$ against $H_1 : \mu_i \neq \mu_j$ for some i and j . By judiciously selecting the size of the test, I can adjust the sensitivity of the test. Clearly if I fail to reject the null, I would believe that all nodes are operating properly. If, however, I reject the null, I would believe that one or more of the nodes is delivering faulty results. Thus I can achieve speedup as well as achieving an indication of faulty nodes.

In this approach, I would want the number of processes, m , on each node to be sufficiently large that CLT approximations would be fairly accurate justifying the use of ANOVA calculations. Thus the sample size also needs to be fairly large. Obviously, this ties in nicely with the need for parallel processing for, if the sample size is not large, there would be no need to parallelize. Clearly the same approach could be used for testing equivalence of medians or testing equivalence of order statistics including maximum and minimum values.

3.2 Stochastic Domain Decomposition for Quadrature

Let me consider the quadrature (numerical integration) of a function, say f . As I described before, I could divide the domain up into strips and, as suggested above, I could systematically assign the strips to each processor. However, if I randomly assign the strips to each processor, then I could reasonably expect the sum of rectangles, $\sum_i(f(X))$ for each $i = 1, \dots, k$ to be approximately the

same where \sum_i is the sum of all strips for the i th node. Again I could use an ANOVA test to examine whether the results for the nodes were statistically equivalent or not.

3.3 Stochastic Domain Decomposition for Multiple Linear Regression

The multiple linear regression application is one of particular interest, which I will examine in more detail. I have selected multiple regression because the procedure is well understood, the computations are straightforward, and the statistical tests of homogeneity are easily developed. The implementation of the parallelization of multiple linear regression was done using a hypercube configured with 16 nodes, a direct routing module for communication via message passing, and an additional vector pipeline coprocessor. In addition to the 16 nodes, there was also a host node, which “directs” the activity of the other nodes. The hypercube system had a distributed, message-passing architecture. Data passes through the host to the nodes and the results are gathered from the nodes back to the host. Given the message-passing nature of the architecture, communication overhead typically plays a significant part in the overall effectiveness of any algorithm.

In general, computational problems which require comparatively little internode communication are the most effective ones on the message passing architectures. Bootstrapping and kernel smoothing operations are examples of computationally intensive tasks that fall into this category. While multiple linear regression is comparatively communications intensive, it does admit a very effective parallel implementation and, moreover, elegantly illustrates our point so that I felt it was quite worth developing. Also it allowed me to investigate the effect of varying the communication packet size on the potential speedup.

I often have the need to study a system in which the changes in several variables may effect the dependent variable. I may know or be willing to assume that the model is expressed as a linear model or I may use a linear model as an approximation to some unknown, more complex model. In either case, least squares estimation yields a computational technique generally known as regression. I distribute the computations necessary for multiple linear regression over several node processors. I then use statistical tests for homogeneity as a redundancy check for hardware and software faults. The tests used in this discussion depend on the assumption of normally distributed residuals for their complete validity although, of course, their nonparametric analogues may also be used. Because my use of these normal tests is as a descriptive statistic to indicate severe deviations from homogeneity, I am not extremely concerned whether the assumption of normality is met exactly or not.

The mathematical model for multiple linear regression can be expressed as follows:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_p x_{ip} + e_i, i = 1, 2, \dots, n$$

or in matrix formulation:

$$\underline{Y} = \underline{1}\beta_0 + \underline{X}\beta_1 + \underline{e},$$

where \underline{Y} is an $(b \times 1)$ vector of observations, $\underline{1}$ is an $(n \times 1)$ vector of ones, β_0 is an unknown parameter, \underline{X} is an $(n \times p)$ matrix of nonstochastic variables, β_1 is a $(p \times 1)$ vector of unknown parameters, and \underline{e} is an $(n \times 1)$ vector of random errors. The traditional assumptions are $E(\underline{e}) = \underline{0}$ and $Cov(\underline{e}) = \sigma^2 \underline{I}$. Thus $E(\underline{Y}) = \underline{X}\beta$ and $Cov(\underline{Y}) = \sigma^2 \underline{I}$. The least squares estimates of β_0 and β_1 , $\hat{\beta}_0$ and $\hat{\beta}_1$, are obtained as follows:

$$\hat{\beta}_1 = (\underline{X}'\underline{X})^{-1}\underline{X}'\underline{Y}, \hat{\beta}_0 = \bar{Y} - \bar{X}'\hat{\beta}_1, \quad (3.1)$$

where $\bar{X} = \underline{X}'\underline{1}/n$ is the vector of column means, $\tilde{X} = \underline{X} - \underline{1}\bar{X}'$ is the centered \underline{X} matrix so that $\tilde{x}_{ij} = x_{ij} - \bar{x}_j$, $\bar{Y} = \underline{Y}'\underline{1}/n$, and $\tilde{Y} = \underline{Y} - \underline{1}\bar{Y}$. I will assume that in the model I work with, I have a nonsingular $\tilde{X}'\tilde{X}$ matrix in each place where I require $(\tilde{X}'\tilde{X})^{-1}$. I may also obtain a Sum of Squared Errors, $SSE = \tilde{Y}'\tilde{Y} - \tilde{Y}'\tilde{X}\hat{\beta}_1$.

To implement multiple linear regression in a parallel fashion, I partition the whole set of the observations and variables into k subsets of close to equal size. I then send each of these subsets to one of the k nodes. I denote data sent to, computed at, or received from node i by adding a subscript (i) to the item. Thus I send to node i : $\underline{Y}_{(i)}$ and $\underline{X}_{(i)}$ of n_i rows each. I compute at node i : $\bar{X}_{(i)}$, $\bar{Y}_{(i)}$, $(\tilde{X}'_{(i)}\tilde{X}_{(i)})$, $(\tilde{X}'_{(i)}\tilde{Y}_{(i)})$, $(\tilde{Y}'_{(i)}\tilde{Y}_{(i)})$, and $SSE_{(i)}$. I note two things at this point: 1) I center at each node. (I use a one pass recursive centering algorithm for speed and accuracy.) 2) I do not compute the slopes and intercepts at each node, although I could if I wished. I am not going to use the node estimates. I merely wish to use the information returned from the nodes to: i) compute the least squares estimates for all the data and ii) assess homogeneity of the results for fault checking.

In order to proceed with my homogeneity checks, I define three potential models for my data. Model 0 is the nominal model. For each node partition of the data, I assume that the slope vector β_1 , and the intercept β_0 are the same. For Model 1, I assume that the node partitions have the same slope vector, but different intercepts. For Model 2, I assume that the node partitions have both different slope vectors and different intercepts. In matrix terms, the three

models are given by:

$$\text{Model 0:} \quad \underset{\sim}{Y}_{(i)} = \underset{\sim}{1}\beta_0 + \underset{\sim}{X}_{(i)}\beta_1 + \underset{\sim}{e}_{(i)}.$$

$$\text{Model 1:} \quad \underset{\sim}{Y}_{(i)} = \underset{\sim}{1}\beta_{0(i)} + \underset{\sim}{X}_{(i)}\beta_1 + \underset{\sim}{e}_{(i)}.$$

$$\text{Model 2:} \quad \underset{\sim}{Y}_{(i)} = \underset{\sim}{1}\beta_{0(i)} + \underset{\sim}{X}_{(i)}\beta_{1(i)} + \underset{\sim}{e}_{(i)}.$$

After aggregating at the host the information computed at the nodes, I proceed to compute a Sum of Squared Errors for each of the three models as follows:

$$\text{Model 2:} \quad SSE_2 = \sum_{i=1}^k SSE_{(i)}.$$

$$\begin{aligned} \text{Model 1:} \quad & (\underset{\sim}{Y}'\underset{\sim}{Y})_{(1)} = \sum_{i=1}^k (\underset{\sim}{Y}'_{(i)}\underset{\sim}{Y}_{(i)}), \\ & (\underset{\sim}{X}'\underset{\sim}{Y})_{(1)} = \sum_{i=1}^k (\underset{\sim}{X}'_{(i)}\underset{\sim}{Y}_{(i)}) \\ & (\underset{\sim}{X}'\underset{\sim}{X})_{(1)} = \sum_{i=1}^k (\underset{\sim}{X}'_{(i)}\underset{\sim}{X}_{(i)}) \\ SSE_1 &= (\underset{\sim}{Y}'\underset{\sim}{Y})_{(1)} - (\underset{\sim}{X}'\underset{\sim}{Y})'_{(1)} (\underset{\sim}{X}'\underset{\sim}{X})_{(1)}^{-1} (\underset{\sim}{X}'\underset{\sim}{Y})_{(1)} \end{aligned}$$

$$\text{Model 0:} \quad n = \sum_{i=1}^k n_i, \quad \bar{\underset{\sim}{X}} = \sum_{i=1}^k n_i \bar{\underset{\sim}{X}}_{(i)} / n, \quad \bar{\underset{\sim}{Y}} = \sum_{i=1}^k n_i \bar{\underset{\sim}{Y}}_{(i)} / n,$$

$$(\underset{\sim}{Y}'\underset{\sim}{Y})_{(0)} = (\underset{\sim}{Y}'\underset{\sim}{Y})_{(1)} + \sum_{i=1}^k n_i (\bar{\underset{\sim}{Y}}_{(i)} - \bar{\underset{\sim}{Y}})^2,$$

$$(\underset{\sim}{X}'\underset{\sim}{Y})_{(0)} = (\underset{\sim}{X}'\underset{\sim}{Y})_{(1)} + \sum_{i=1}^k n_i (\bar{\underset{\sim}{X}}_{(i)} - \bar{\underset{\sim}{X}}) (\bar{\underset{\sim}{Y}}_{(i)} - \bar{\underset{\sim}{Y}})$$

$$(\underset{\sim}{X}'\underset{\sim}{X})_{(0)} = (\underset{\sim}{X}'\underset{\sim}{X})_{(1)} + \sum_{i=1}^k n_i (\bar{\underset{\sim}{X}}_{(i)} - \bar{\underset{\sim}{X}}) (\bar{\underset{\sim}{X}}_{(i)} - \bar{\underset{\sim}{X}})'$$

$$SSE_0 = (\underset{\sim}{Y}'\underset{\sim}{Y})_{(0)} - (\underset{\sim}{X}'\underset{\sim}{Y})'_{(0)} (\underset{\sim}{X}'\underset{\sim}{X})_{(0)}^{-1} (\underset{\sim}{X}'\underset{\sim}{Y})_{(0)}$$

I calculate the regression solutions using equation (3.1) with the summary statistics computed for Model 0. The degrees of freedom associated with the Error Sums of Squares are given by $df_0 = n - p - 1$, $df_1 = n - p - k$, $df_2 = n - kp - k$.

I may now calculate two test statistics. I may test for Total Homogeneity (which is the true redundancy test for fault checking) and for Homogeneity of Slopes Only (which I have included simply because it is so easy to do and might provide some detail about what went wrong if something did). The test for Total Homogeneity uses the statistics: $SS_{(2,0)} = SSE_0 - SSE_2$, $df_{(2,0)} = df_0 - df_2 = (k - 1)(p + 1)$, $MS_{(2,0)} = SS_{(2,0)} / df_{(2,0)}$, $F_{(2,0)} = MS_{(2,0)} / MSE_2$, where $MSE_2 = SSE_2 / df_2$.

The test for Homogeneity of Slopes Only uses the statistics: $SS_{(2,1)} = SSE_1 - SSE_2$, $df_{(2,1)} = df_1 - df_2 = (k - 1)(p + 1)$, $MS_{(2,1)} = SS_{(2,1)}/df_{(2,1)}$, $F_{(2,1)} = MS_{(2,1)}/MSE_2$, where $MSE_2 = SSE_2/df_2$. If heterogeneity is detected, then further tests may be made to isolate the nodes with different and presumed faulty results. I note at this point that I set the significance level for my homogeneity test very small. This is because I want a very small false alarm rate. I only want to detect egregious deviations from homogeneity, as might be caused by a hardware or software failure.

The methodology described above is designed to isolate potentially catastrophic failures in the node hardware or software. However, it could also be used to affect a speedup of other kinds of homogeneity checks on data. For example, suppose that instead of parsing the data to allocate it to nodes in a manner which creates random samples, I allocated the data to correspond to some meaningful partition of the data such as orthants of the \tilde{X} space. The parallel algorithm described above would then yield a speedup of this homogeneity check, but would no longer have any fault detection capability. However, a simple modification whereby I split each partition into two or more subpartitions via random sampling would still give a homogeneity check using straightforward extensions of the above methodology.

3.3.1 Timing Results

The timing study was designed to measure the effectiveness of the parallel scheme described above, and, in particular, to measure the effect of changing the size of the communications packets sent between the host and the nodes. I began by generating data files of similar data. I did this by taking an original data set with six independent variables and then generating data sets of arbitrary size. I then matched the covariance structure of the rows of the \tilde{X} matrix with that of the original problem, made the regression coefficients the same as in the original problem, and matched the variability of the generated residuals with those of the original problem. Hence, regardless of the size of the test data set, I could be assured that it stochastically agreed with the original data set. In this sense, my test data sets were comparable. The sizes selected for this part of the study were $n = 8000$ and 16000 observations. I also used various numbers of nodes, so that the speedup from parallelizing could be determined.

The study also measured the effect of differing sizes of communication packets sent from the host to the nodes. The sizes used in this study were 125, 250, and 500 observations per node. Since I used a “broadcast” transmission of the data for all nodes, with each node picking its data out of the message, the size of packet transmitted also depends on the number of nodes. The size of the actual transmitted packet is number of nodes times package size. It might appear that one should automatically choose the largest possible size of packet in order to minimize the effect of communications start up overhead. However, this could lead to nodes remaining idle while transmission is taking place. Hence, it may be more effective to use smaller packets, so that the nodes may continue doing

productive work. I used several sizes to examine the effect of package size on overall efficiency.

I measured at each node the overall time for the program, the time waiting for the host to read data, the computation time, and data transmission time (both sending and receiving). My measure of effective time for the computations is the maximum over nodes of overall time minus time waiting for the host to read the data. Hence, the computation time and the communications overhead time for the hypercube are included in my time measure, but the time for the host to initially read in the data is not included. The speedup for any given number of nodes for a particular configuration is given by the ratio of the time for one node divided by the time measure for that number of nodes. Times were measured by an internal clock subroutine on the hypercube and are given in milliseconds. The results of my simulations are given in Table 1. Each number is the average for two runs.

I observe from Table 1 that the effective times indeed decrease as I add more nodes. I also note, as is well known, that the speedup is not linear in the number of nodes. The speedups achieved for the six rows of Table 1 (from one node to sixteen) are respectively: 12.65, 13.55, 11.79, 13.16, 10.43, and 12.31. The speedups are greater for the larger data set and are greater for package size 125 observations per node than for the larger packages. The reason that speedup is not perfectly linear is that communications overhead increases as the number of nodes increases. However, as might be expected for perfectly parallel computations as I have here, the computation time indeed decreases as the reciprocal of the number of nodes. In fact, a regression of computation time (again the maximum over nodes) versus sample size divided by number of nodes yields an R^2 of 0.999978. The communication overhead prevents achieving perfect linear speedup.

I also made some additional runs with larger sample sizes to explore the limiting behavior of the speedup. I wished to observe where the asymptote, if any was with respect to increased speedup and sample size. Hence, I made additional runs with one and sixteen nodes for each package size for sample sizes 32000, 64000, and 128000. The results of these runs and some information from Table 1 are presented in Table 2.

As may be seen from Table 2, the speedup appears to have an asymptotic value of approximately 13.8 for sixteen nodes. This seems to be the case regardless of the package size. Nevertheless, for any given sample size, the smaller package size gives smaller effective times and larger speedup. Hence, I observe the phenomenon that the desired efficiency of the larger package size (namely, the lesser number of times the communication startup overhead is involved) is overcome by the fact that the nodes sit idle waiting for data to arrive with the larger package sizes.

I make one final remark with regard to effective time. If the time at the host for reading in the data is included, the time to read in the data overwhelms the computations for this problem. I conceive of a situation in which the data are acquired in some automated mode which can bypass the reading step that I did here. This is reasonable, since the fault checking feature described above

would be critical in a situation where the data arrived in huge amounts and was processed in an automated fashion. The effective time as I have measured it gives a fair reading of the speedup from the parallel implementation of regression. All communication overhead is included except the reading of the data. This is a commonly applied method for measuring speedup.

Again I use the maximum of the node times as my measure of the node processing time. Time is in milliseconds and each time is for one run. The results for 1, 2, 4, 8, and 16 nodes were respectively: 8,292,879; 4,143,741; 2,082,504; 1,055,863; and 545,479. The speedup from one to sixteen nodes was 15.20. The parallelism achieves close to perfect linear speedup in the number of nodes. The communication time is extremely small compared to the large amount of computation time for this application.

4 Stochastic Load Balancing

In the discussion of stochastic domain decomposition, I inherently made the assumption that I were dividing the data into equal size pieces. This is done so that each node will have the same amount of data to work on and hence finish the computation in approximately the same amount of time. For linear algorithms such as such as computation of means, variances, kernel density estimators, quadrature, and multiple linear regression as discussed above this is a reasonable assumption and I could expect to achieve an approximately balanced load. However, if the algorithms are adaptive and/or the nodes are not necessarily of the same capability, then load balancing becomes a much more complex issue. A parallel computation is only as fast as the slowest node.

Thus in the former case, a natural and obvious procedure in a k -node SIMD machine is to partition the data/algorithm into k equivalent pieces and send each of the k pieces to a node. In a perfectly deterministic machine, all nodes would complete their computations simultaneously and return their completed computations to the controller node. Actually, the simultaneous completion of the computations would be undesirable since the nodes would be simultaneously contending for communication channels or buses. Nonetheless, large discrepancies in completion time can significantly extend computation times. These discrepancies may arise because the execution time of an algorithm is data dependent and/or because the processing time at each node varies. As an example in the former case, consider a numerical integration routine whose speed of convergence depends on the gradient of the integrand. The natural partition might send some nodes parts of the integrand where the gradient is large and other nodes parts where the gradient is smaller. Completion times differing by a factor of two or more are not unreasonable.

Even when data differences are not a significant factor, computation times at each node may differ significantly. For example, in concurrent computers consisting of independent processors linked by communication channels, the impact of contention for communication channels may have significant effect. A processor node may complete a partial computation on the order of microseconds,

but wait on the order of milliseconds for access to the communication linkage. Thus a difference of a few microseconds could conceivably cause one node to win the contention for a communication over another node and substantially alter the time to completion. Even when contention for a communications link is not an issue, there may be substantial differences among performances of nodes due to environmental and other factors. The fundamental theme is that for a variety of reasons performance of nodes of a parallel computer with otherwise identical algorithms is subject to statistical fluctuation. This fluctuation causes an imbalance in the load on the various nodes and consequently will affect the total computation time adversely. In view of the stochastic character of the load imbalance, I construct load balancing algorithms designed to minimize overall computation times. The following discussion is based on Wegman [8]

4.1 A Stochastic Model

I seek a stochastic algorithm for decomposing a computational task among a number of processors. In the first model, I assume a basic computational task of size, W . I leave the meaning of computational task undefined, but I have in mind a segment of code or an algorithm and the associated data. I assume that I have k processors and that I may decompose task W into subtasks of size W_j , $j = 1, \dots, k$. Let T_j be the computation time for the j th task. I generally have in mind parallel computers with a relatively small number of nodes. Such architectures as the hypercube or shared memory machines with 16 to say 128 nodes are examples. This model would also serve for a distributed computing environment consisting of a network of workstations. In order to construct the basic model, I make a few simplifying assumptions.

- a. Any task can be partitioned arbitrarily finely.
- b. Computation time at any node is directly proportional to task size.
- c. Communication time is negligible.
- d. T_1, \dots, T_k are independent random variables with common density $f(t)$.

A very simple decomposable task is one or more nested DO loops. Obviously, the finest partition I can construct is to the level of the individual sequence of code in the innermost loop. The assumption a is obviously unrealistic taken to the limit. However, in any application demanding high performance computing, the size of the loop will be very much bigger than the number of processors and the discretization approximation will be negligible. Assumption b is simply a linearity assumption and seems reasonable when applied to the nodes individually. Assumption c is the most problematic. Certainly, in architectures such as the hypercube or networks of workstations, which are communication intensive, this is hardly a realistic assumption for many types of tasks. Nonetheless, I have in mind applications where the tasks are computationally very intensive (say minutes or hours at each node) and the communication traffic is negligible

while the nodes are operating. It should be clear that if I am dealing with tasks whose basic computation time at each node is on the order of seconds or less, load balancing is something of a moot issue. Finally, assumption d is a statistical assumption most appropriate in the SIMD framework. The model discussed here may be extended to the case where non-identical tasks are sent to distinct nodes (i.e., the MIMD framework).

To develop the model, I let $T_{(k)} = \max\{T_1, \dots, T_k\}$. The complete task W will take time $T_{(k)}$ to compute on a k -node processor. Observe that $T^* = \sum_{j=1}^k (T_{(k)} - T_j)$ is the total idle time. The statistical distribution of $T_{(k)}$ is

$$g(t) = k[F(t)]^{k-1}f(t), t > 0 \quad (4.1)$$

where $F(t)$ is the distribution corresponding to density, $f(t)$. Our approach to load balancing in general is to use the idle time, T^* , by creating a nondegenerate partition of W_j into pieces X_j and Y_j so that $T_j = U_j + V_j$ where U_j is the computation time for X_j , V_j is the computation time for Y_j . The procedure is to start processor j with task X_j , the Y_j 's being held as load balancing tasks. When a processor is finished with its current task, it is assigned one of the Y_j filler tasks.

At issue is to determine the balance between U_j and V_j . Several criteria can be proposed.

a.
$$\sum_{j=1}^k EV_j = \sum_{j=1}^k E(U_{(k)} - U_j) = k (EU_{(k)} - EU_j)$$

In this case, all the filler tasks are expected to be computed in the expected idle time. Hence total expected compute time is $EU_{(k)}$ where $U_{(k)} = \max\{U_1, \dots, U_k\}$. Since $V_j > 0$ with probability one, I have $U_j < T_j$ with probability one so that $U_{(k)} < T_{(k)}$ with probability one. It follows that $EU_{(k)} < ET_{(k)}$.

b.
$$E \max V_j = EU_{(k)} - EU_{(1)},$$

In this case, the right hand side is the longest expected idle time for any processor. The left hand side corresponds to the expected computing time for the load balancing task Y_j requiring the longest computing time. Here $U_{(1)} = \min U_j$.

c.
$$EV_j = EU_{(k)} - EU_{(1)}.$$

Similar to criterion b, but the left hand side corresponding to the expected computing time for the average load balancing task Y_j rather than the worst case situation. Clearly this is a more conservative criterion.

These criteria may be implemented in the following way. Let $U_j = \theta T_j$ and $V_j = (1 - \theta)T_j$. Then criterion a becomes

$$\text{a}^*. \quad n (1 - \theta) ET_j = n \theta (ET_{(k)} - ET_j)$$

or

$$\theta_a = ET_j / ET_k.$$

Similarly, criterion b becomes

$$\text{b}^*. \quad (1 - \theta) ET_{(k)} = \theta ET_{(k)} - \theta ET_{(1)}$$

or

$$\theta_b = \frac{ET_{(k)}}{(2ET_{(k)} - ET_{(1)})}.$$

Finally, criterion c becomes

$$\text{c}^*. \quad (1 - \theta) ET_j = \theta ET_{(k)} - \theta ET_{(1)}$$

or

$$\theta_c = \frac{ET_j}{(ET_j + ET_{(k)} - ET_{(1)})}.$$

Criteria a* through c* give explicit values for the load balancing parameter θ in terms of parameters, which may be estimated from knowing the probability density $f(t)$ associated with distribution of computation times T_j , in particular, in terms of moments of the random variables T_j or their order statistics. Notice that if μ and σ are respectively the mean and standard deviation associated with T_j , then as $\mu/\sigma \rightarrow \infty$,

$E \max T_j = ET_{(k)} \rightarrow ET_j$ and $E \min T_j = ET_{(1)} \rightarrow ET_j$ so that θ_a , θ_b and $\theta_c \rightarrow 1$. Thus, as I might hope for the deterministic case ($\sigma = 0$), $\theta = 1$ and no load balancing partitioning is necessary.

I close this section by observing that $\sum_i EU_i$ is the expected compute time for the X portion of the task while $\sum_i EV_i$ is the expected compute time for the Y portion of the compute task. However, $\sum_i (EU_{(k)} - EU_i)$ is the expected idle time which the Y tasks can use as filler. Thus the total expected computation time is $\sum_i EU_i + \sum_i EV_i - \sum_i (EU_{(k)} - EU_i) = k(ET_i + EU_i - EU_{(k)})$. Since kET_i is the expected computation time with no stochastic load balancing effort,

$k(EU(n) - EU_i) = \theta ET^*$ is the expected reduction in computing time due to stochastic load balancing.

4.2 Moment Calculations

I calculate the moments for a number of simple cases. Perhaps the simplest reasonable model for the distribution $F(t)$ of computation times T_j is the exponential distribution. For numerical comparison, I also calculate the appropriate moments of the uniform.

4.2.1 Exponential Distribution

Consider the exponential density

$$f_\lambda(t) = \lambda^{-1} e^{-t/\lambda}, t > 0$$

with distribution

$$F_\lambda(t) = 1 - e^{-t/\lambda}, t > 0.$$

From (4.1) I know that $T_{(k)} = \max\{T_1, \dots, T_n\}$ has density

$$g_\lambda^{(k)}(t) = k [F_\lambda(t)]^{k-1} f_\lambda(t), t > 0.$$

It follows immediately that

$$G_\lambda^{(k)}(t) = [F_\lambda(t)]^k, t > 0.$$

I may calculate the expectation by

$$ET_{(k)} = \int_0^\infty [1 - G_\lambda^{(k)}(t)] dt = \int_0^\infty [1 - (1 - e^{-t/\lambda})^k] dt.$$

Let $x = 1 - e^{-t/\lambda}$ so that $dt = \frac{\lambda dx}{1-x}$. Thus

$$ET_{(k)} = \int_0^1 (1 - x^k) \frac{\lambda dx}{(1-x)}$$

which yields

$$ET_{(k)} = \int_0^1 (1 + x + x^2 + \dots + x^{k-1}) \lambda dx$$

or

$$ET_{(k)} = \lambda \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}\right). \quad (4.2)$$

Similarly, the distribution of $T_{(1)} = \min T_j$ is given by

$$G_\lambda^{(1)}(t) = 1 - [1 - F_\lambda(t)]^k, t > 0.$$

The expectation is

$$ET_{(1)} = \int_0^\infty [1 - G_\lambda^{(1)}(t)] dt = \int_0^\infty e^{-tk/\lambda} dt = \lambda/k. \quad (4.3)$$

Finally, the expected value of T_j is

$$ET_j = \lambda. \quad (4.4)$$

Formulae (4.2) through (4.4) allow me to compute the value of θ_i for $i = a, b, c$. In the case of Criteria a through c, the solutions in the exponential case are

$$\begin{aligned} a^\dagger. \quad & \theta_a = \frac{1}{1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}}. \\ b^\dagger. \quad & \theta_b = \frac{1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}}{2(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}) - \frac{1}{k}}. \\ c^\dagger. \quad & \theta_c = \frac{1}{2 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k-1}}. \end{aligned}$$

Table 3 gives some values of the load balance parameter for various number of nodes. Notice that there is no dependency on the parameter λ of the exponential distribution. Notice also the criteria a and c both yield negative values of the load balance parameter indicating that these criteria are inappropriate with the exponential distribution. The exponential is in some sense a pathological distribution. It is frequently used as a memoryless distribution, i.e. given that an item has not yet failed, it will have the same probability of failure as when it was new. Thus in the load balancing context, given that a computation is not yet finished, the probability distribution associated with the time to completion is the same as when it first began. The gamma family of distributions generalizes the exponential and perhaps provides some more realistic models for computation times. However, because of its more complicated form, computation of load balance parameters in closed form are not readily available.

4.2.2 Uniform Distribution, Case 1

A second illustrative example is the uniform density. The density uniform on $(0, m)$ is given by

$$f_m(t) = 1/m, 0 < t < m$$

with distribution

$$F_m(t) = t/m, 0 < t < m.$$

Again from (4.1) the distribution of $T_{(k)}$ has density

$$g_m^{(k)}(t) = \frac{k}{m^k} t^{k-1}, 0 < t < m.$$

From this it follows that

$$ET_{(k)} = \int_0^m \frac{k}{m^k} t^k dt = \frac{km}{k+1}. \quad (4.5)$$

The density of $T_{(1)}$ is

$$g_m^{(1)}(t) = \frac{k}{m} \left(1 - \frac{t}{m}\right)^{k-1}, 0 < t < m.$$

A simple calculation shows that

$$ET_{(1)} = \int_0^m \frac{k}{m} \left(1 - \frac{t}{m}\right)^{k-1} t dt = \frac{m}{k+1}. \quad (4.6)$$

Finally for a uniform density $ET_j = m/2$. Thus using (4.5) and (4.6) I may again calculate the load balance parameters for criteria a through c.

$$\text{a}^\ddagger. \quad \theta_a = \frac{k+1}{2k}.$$

$$\text{b}^\ddagger. \quad \theta_b = \frac{k}{2k-1}.$$

$$\text{c}^\ddagger. \quad \theta_c = \frac{k+1}{3k-1}.$$

Table 4 gives some values of the load balance parameters in the case of uniform densities. Notice again that the load balance parameter is independent of m . The reason for this is perhaps a bit subtle. It should be noted that in all of the load balance criteria a through c, the left hand side of the load balance equation is a location dependent quantity while the right hand side is a scale dependent quantity. In both distribution examples, the location and scale parameters are both proportional to the parameters λ and m respectively. Hence, the parameters themselves wash out. In particular, $\mu/\sigma = 1$ for the exponential case and $\mu/\sigma = \sqrt{3}$ for the uniform case. Both of these are comparatively small values and far from ∞ . In the next example, I consider a situation where location and scale are not proportional.

4.2.3 Uniform Distribution, Case 2

In this third example I again consider the uniform case, but this time with density given by

$$f_m(t) = 1, m-1 < t < m.$$

In this case, it is straightforward to show that

$$ET_{(k)} = m - \frac{1}{k+1},$$

$$ET_{(1)} = m - \frac{k}{k+1},$$

and

$$ET_j = m - \frac{1}{2}.$$

Simple calculations show that

$$a'. \quad \theta_a = \frac{m - \frac{1}{2}}{m - \frac{1}{k+1}},$$

$$b'. \quad \theta_b = \frac{m - \frac{1}{k+1}}{m + \frac{k-2}{k+1}},$$

$$c'. \quad \theta_c = \frac{m - \frac{1}{2}}{m + \frac{k-1}{k+1} - \frac{1}{2}}.$$

Table 5 contains load balance parameters for this example with $m = 3$. In this case $\mu/\sigma = 5\sqrt{3}$. Tables 6 through 8 contain calculations of the expected reduction in computing time for the three examples considered.

4.3 Randomized Load Balance Parameters

In the previous section, I consider the case where θ was a fixed, but unknown number which could be calculated from the distribution associated with T_j . In this section I consider θ_j to be a random variable with conditional density $g_\nu(\theta | t_j)$ where as before $U_j = \theta_j T_j$. The joint density of θ_j and T_j is given by $g_\nu(\theta | t) f(t)$. Letting $Z_j = T_j$, I have a jacobian of 1 and it follows that the joint density of U_j and Z_j is

$$h_\nu(u, z) = h_\nu\left(\frac{u}{z} \mid z\right) f(z)$$

and that the marginal density of U_j is

$$h_\nu(u) = \int h_\nu\left(\frac{u}{z} \mid z\right) f(z) dz.$$

Using the formulae developed earlier, I calculate total expected computation time. Since T_j is a positive random variable a reasonably rich family of plausible distribution is the gamma family, i.e.

$$f_{\gamma,\delta}(t) = \frac{1}{\Gamma(\gamma)\delta^\gamma} t^{\gamma-1} e^{-\frac{t}{\delta}}, t > 0.$$

Also since I desire θ between 0 and 1, a reasonable family for θ is the beta.

$$g_{\alpha,\beta}(\theta) = B(\alpha,\beta)\theta^{\alpha-1}(1-\theta)^{\beta-1}, 0 < \theta < 1.$$

Hence the joint density of U_j and Z_j is

$$h_{\alpha,\beta,\gamma,\delta}(u, z) = \frac{B(\alpha,\beta)}{\Gamma(\gamma)\delta^\gamma} \left(\frac{u}{z}\right)^{\alpha-1} \left(1 - \frac{u}{z}\right)^{\beta-1} z^{\gamma-1} e^{-\frac{z}{\delta}}, u, z > 0.$$

The parameters α and β may depend on $Z_j = t_j$. Hence the marginal density of U_j may be written as

$$h_{\gamma,\delta}(u) = \frac{1}{\Gamma(\gamma)\delta^\gamma} \int_0^\infty B(\alpha,\beta) u^{\alpha-1} (z-u)^{\beta-1} z^{\gamma-\alpha-\beta+1} e^{-\frac{z}{\delta}} dz. \quad (4.7)$$

One possible choice for α and β is $\alpha = z/(1+z)$ and $\beta = 1/(1+z)$. In this case I have

$$h_{\gamma,\delta}(u) = \frac{1}{\Gamma(\gamma)\delta^\gamma} \int_0^\infty B\left(\frac{z}{1+z}, \frac{1}{1+z}\right) u^{-\frac{1}{1+z}} (z-u)^{-\frac{z}{1+z}} z^\gamma e^{-\frac{z}{\delta}} dz. \quad (4.8)$$

The density of V_j can be calculated in a similar manner by replacing θ with $1-\theta$, that is to say interchanging the roles of α and β . The procedure then is to choose a value of θ_j according to g_ν . If I have some experience with the distribution of the T_j , I may make my choice dependent on the T_j 's. I have in mind here potentially choosing θ_j larger (hence smaller filler tasks) to be used toward the end of the run. In any case, the conditional density of θ on t_j need not actually depend on t_j . It should be clear from the rather complicated formulae

(4.7) and (4.8) that a closed form expression for the densities of U_j and of V_j is not readily feasible. Nonetheless an expression for EU_j can be found as

$$EU_j = \int_0^\infty u h_{\gamma,\delta}(u) du.$$

That is

$$EU_j = \int_0^\infty \int_0^\infty \frac{1}{\Gamma(\gamma)\delta^\gamma} B(\alpha,\beta) u^{\alpha+1-1} (z-u)^{\beta-1} z^{\gamma+1-(\alpha+1)-\beta+1} e^{-\frac{z}{\delta}} du dz.$$

So that

$$EU_j = \frac{\Gamma(\gamma+1)B(\alpha,\beta)\delta}{\Gamma(\gamma)B(\alpha+1,\beta)}.$$

By symmetry

$$EV_j = \frac{\Gamma(\gamma+1)B(\beta,\alpha)\delta}{\Gamma(\gamma)B(\beta+1,\alpha)}.$$

Unfortunately the computation of the expectation of $U_{(n)}$ is considerably more difficult so that numerical methods must be pursued.

5 Summary and Conclusions

High performance computing has historically been the realm of those who have engaged in mathematical modeling using partial differential equations. The recent focus on data mining of massive data sets has focused the interest of statisticians and related fields on the possibility, indeed, the necessity of focusing on high performance parallel computation. The interaction between high performance parallel computing and statistical methods is a two-way street. I have tried to illustrate this with the discussion of stochastic domain decomposition and stochastic load balancing. In the former, parallel computing offers the possibility of speedup of many different types of statistical algorithms while the stochastic component allows for an ability for fault detection, an ability not present in the usual domain decomposition methods. With the stochastic load balancing methodology, knowledge of statistical methods suggests an approach to improving performance of parallel computing. This two-way street should encourage more statisticians and computer scientists to interact.

Acknowledgments

This work was completed under the sponsorship of the Office of Naval Research under contract DAAD19-99-1-0314 administered by the Army Research Office, by the Air Force Office of Scientific Research under contract F49620-01-1-0274 and contract DAAD19-01-1-0464, the latter also administered by the Army Research Office and finally by the Defense Advanced Research Projects Agency

through cooperative agreement 8105-48267 with the Johns Hopkins University.

References

- [1] E. J. Wegman. Huge data sets and the frontiers of computational feasibility. *Journal of Computational and Graphical Statistics* 4(4):281-295, 1995
- [2] W. Gropp and E. Lusk. Implementing MPI: the 1994 implementors' workshop. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, Mississippi, October, 1994
- [3] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface (2nd ed.)*. Cambridge, MA: The MIT Press, 1999
- [4] M. Snit, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, The MPI Core (Vol. I, 2nd ed.)*. Cambridge, MA: The MIT Press, 1998
- [5] J. Dongarra, I. Duff, P. Gaffney, and J. McKee. *Vector and Parallel Computing: Issues in Applied Research and Development*. New York, NY: John Wiley and Sons, Inc., 1989
- [6] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Reading, MA: Addison-Wesley Publishing Company, 1995
- [7] M. Xu, E. J. Wegman, and J. J. Miller. Parallelizing multiple linear regression for speed and redundancy: an empirical study. *Journal of Statistical Computation and Simulation*. 39: 205-214, 1991
- [8] E. J. Wegman. A stochastic approach to load balancing in coarse grain parallel computers. In *Computing and Graphics in Statistics*, (A. Buja and P. Tukey, eds.) pp.219-230, New York: Springer-Verlag, 1991

Table 1: Results of Part 1 of the Timing Study

Observations per node per package	Sample Size	Effective Time				
		Number of Nodes				
		1	2	4	8	16
125	8000	15337	7722	3931	2058	1212
	16000	30703	15458	7854	4088	2266
250	8000	15337	7721	3935	2084	1301
	16000	30699	15448	7852	4106	2332
500	8000	15358	7733	3957	2155	1472
	16000	30737	15467	7877	4155	2496

Table 2: Results of Part 2 of the Timing Study

Observations per node per package	Sample Size	Effective Time		
		Number of Nodes		
		1	16	Speedup
125	8000	15337	1212	12.65
	16000	30703	2266	13.55
	32000	61471	4460	13.78
	64000	122132	8855	13.79
	128000	243648	17667	13.79
250	8000	15337	1301	11.79
	16000	30699	2332	13.16
	32000	61445	4521	13.59
	64000	122199	8905	13.72
500	128000	243714	17685	13.78
	8000	15358	1472	10.43
	16000	30737	2496	12.31
	32000	61515	4661	13.20
	64000	122309	9043	13.53
	128000	244085	17815	13.70

Table 3: Load Balance Parameters for the Exponential Distribution

k	θ_a	θ_b	θ_c
1	1.000	1.000	1.000
2	.667	.800	.500
3	.545	.720	.406
4	.480	.676	.353
8	.368	.602	.278
16	.296	.559	.232
32	.246	.539	.199
64	.211	.519	.174
128	.184	.511	.156
∞	.000	.500	.000

Table 4: Load Balance Parameters for the Uniform Distribution (1)

k	θ_a	θ_b	θ_c
1	1.000	1.000	1.000
2	.750	.667	.600
3	.667	.600	.500
4	.625	.571	.455
8	.563	.533	.391
16	.531	.516	.362
32	.516	.508	.347
64	.508	.504	.340
128	.504	.502	.337
∞	.500	.500	.333

Table 5: Load Balance Parameters for the Uniform Distribution (2)

k	θ_a	θ_b	θ_c
1	1.000	1.000	1.000
2	.938	.889	.882
3	.909	.846	.833
4	.893	.824	.806
8	.865	.788	.763
16	.850	.769	.739
32	.842	.760	.727
64	.838	.755	.720
128	.835	.752	.717
∞	.833	.750	.714

Table 6: Expected Reduction in Computing Time for the Exponential

k	a	b	c
1	.000	.000	.000
2	.667	.800	.500
3	1.364	1.800	1.000
4	2.080	2.930	1.529
8	5.056	8.277	3.825
16	11.267	21.296	8.821
32	24.115	52.249	19.468
64	50.509	124.416	41.829
128	104.441	289.873	88.313

Table 7: Expected Reduction in Computing Time for the Uniform (1)

k	a	b	c
1	.000	.000	.000
2	.250	.222	.200
3	.500	.450	.375
4	.750	.686	.545
8	1.750	1.659	1.217
16	3.750	3.643	2.553
32	7.750	7.634	5.721
64	15.750	15.630	10.555
128	31.750	31.627	21.222

Table 8: Expected Reduction in Computing Time for the Uniform (2)

k	a	b	c
1	.000	.000	.000
2	.313	.296	.294
3	.682	.625	.625
4	1.071	.988	.968
8	2.692	2.451	2.373
16	6.000	5.430	5.217
32	12.650	11.418	10.925
64	25.979	23.412	22.350
128	52.642	47.409	45.206