How to Choose the Grain Size of a Parallel Computer

Donald Yeung, William J. Dally, and Anant Agarwal
MIT Laboratory for Computer Science and
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Designers of parallel computers have to decide how to apportion a machine's resources between processing, memory, and communication. How these resources are apportioned determine the grain and balance of the resulting machine. Often, these design decisions are made according to rules of thumb which often lead to unoptimized designs. This paper presents an analytical framework upon which designers can reason about the design space of parallel computers, and make design decisions based on analysis. The framework is grounded upon the notion of cost-effective design. It focuses on the needs of applications and tries to identify machines that will execute these applications efficiently. This type of optimization is made difficult by the fact that the application domain may be diverse, each application demanding very different resources for efficiency.

This paper makes three contributions in the context of parallel computer design. First, it provides an analytical framework based on a "blc mpP" machine characterization that considers processing, memory, local and global communication, and latency as separate machine resources. This characterization is unique because it accounts for locality by considering local and global communication separately. Second, the model predicts that general-purpose parallel computers are realizable, but may be possible only at extremely high machine budgets. Finally, the model shows that the amount of memory architected into current production parallel machines is suited for machines with 10s to low 100s of processors. These machines are not cost-effective for moderately or massively parallel systems unless astronomical problem sizes are desirable.

1 Introduction

A parallel computer consists of a set of nodes interconnected by a network. Each node is comprised of processor(s), memory, and communications as shown in Figure 1. All parallel computers share this structure regardless of the execution model they support: shared-memory, message-passing, or dataflow. Parallel computers are distinguished by the mechanisms they provide to support their execution model and by the division of resources across the nodes and within each node.

Two key questions in the design of a parallel computer involve grain size (the size of each node) and balance (the relative size and performance of the components within each node). Grain size and balance play a large part in determining the efficiency or performance per unit cost of a machine. If an engineer builds a small number of very large nodes, a point of diminishing returns is reached where node performance increases very slowly (if at all) as node size is increased. On the other hand, building a large number of very small nodes will also result in diminishing returns

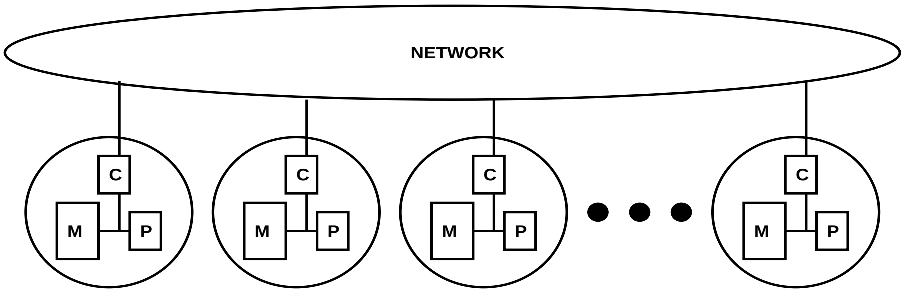


Figure 1: All parallel computers consist of a set of nodes interconnected by a network. Each node consists of processor(s), memory, and communications.

as processor performance drops and communication costs rise. The highest efficiency occurs at an optimal point between the two extremes. Similarly, as observed by Kung [4], there is an optimal balance of resources between the processor, memory, and communication components within a node.

In this paper, we develop an analytical framework for choosing the grain size and balance of a parallel computer that is based on a "blc mpP" (pronounced "block MPP") machine characterization. We choose as independent variables the number of nodes, P, processing power per node, p, the memory per node m, and the communication bandwidth per node, c. As an extension to the model, we consider global bandwidth per node, b, and network latency, l. Using these variables, we derive formulae for performance and cost.

A performance function for an application estimates the running time of the application as a function of the variables. An ensemble performance function is used to characterize the performance across a set of applications. Cost functions estimate the cost of realizing a parallel computer with a given set of variables. We approximate the cost function of the whole computer as the sum of the cost functions of its components and independently derive cost functions for the processor, memory, and communications components. Using cost functions, we predict how the cost of a computer changes as we vary the number of nodes and the composition of each node.

Together, the cost functions and performance functions predict the efficiency (performance/cost) of a machine and thus allow us to perform a number of constrained optimizations on our independent variables. We can, for example, compute the points and contours in *machine space* that correspond to best efficiency, best performance for a given cost, and lowest cost for a given level of performance.

The current trend to base the grain size and balance of multicomputer nodes on the size and balance of current workstations while convenient usually results in machine configurations that are far from the optimal contour. While there has been much debate on this topic, few concrete results have been reported. Machine balance continues to be determined more by convenience and market forces than by engineering analysis. Our primary motivation in undertaking this study is to provide an analytical framework to enable engineers to logically choose machine grain size and balance.

This study calls into question the notion of a *general-purpose* machine. A machine configuration tuned to a particular application may have much greater efficiency on that application than a general purpose machine that must make compromises to perform well on a broad range of applications. Thus, just as conventional machines are often configured with different amounts

Machine Parameters	
p	Processing power per node (operations/cycle).
m	Memory size per node (words).
c	Communication bandwidth per node (words/cycle).
b	Global communication bandwidth per node (words/cycle).
l	Communication latency per node for zero-length global message (cycles).
P	Number of nodes.
V	A machine configuration vector: P, p, m, c, b, l.
Cost Parameters	
$K(V)$	Cost of a given machine configuration, V (in DRAM bit equivalents or Dbe).
$K_n(p, m, c)$	Cost of a node with configuration, (p, m, c) (Dbe).
$K_m(m)$	Cost of memory with capacity m (Dbe).
$K_p(p)$	Cost of a processor with performance p (Dbe).
$K_c(c)$	Cost of communications with bandwidth c (Dbe).
$K_b(b)$	Cost per node of global communication with bandwidth b (Dbe).
$K_l(l)$	Cost of supporting communication latency l per node (Dbe).
Application Parameters	
N	problem size
$R(N, P)$	Requirements vector for an application.
$R_p(N, P)$	Required number of processing operations per node.
$R_m(N, P)$	Required amount of memory words per node.
$R_c(N, P)$	Required number of words of local communication per node.
$R_b(N, P)$	Required number of words of global communication per node.
$R_l(N, P, V)$	Latency inherent to the computation.

Table 1: Basic Model parameters for machine, cost, and applications.

of processing, memory, and I/O to support commercial versus scientific applications, we can expect parallel computers to be configured with different grain sizes and different balances for different application areas.

The remainder of this paper explores the issues of grain size and balance in more detail. Section 2 describes the notation used throughout the paper. Section 3 gives a qualitative analysis of cost and performance. Section 4 develops a simple cost model, and Section 5 develops a simple performance model. These two simple models are extended to include the effects of global bandwidth and latency in Section 6. Section 7 discusses solutions to both the simple and extended models, and in Section 8, we present the results of these solutions. Finally, in Section 9, we present our conclusions. s.

2 Nomenclature

Table 1 the notation used in this paper. It points out the three important categories of parameters for our model: machine, cost, and application parameters. Throughout the paper, performance and execution time are measured in units of machine *cycles*, information in units of machine *words*, and cost in *DRAM bit equivalents* which are defined in Section 4. Refer to Appendix A for an exhaustive list of notation.

3 Qualitative Analysis

It is instructive to examine the qualitative properties of the cost and performance functions before deriving specific formulae.

If we assume that these functions are analytic and monotonic, we can make some qualitative observations about the following machine configurations:

1. V_e, the vector of values, P, p, m, c, b, l, that gives the highest overall efficiency on a class of applications.
2. $V_k(k)$, the vector of values that gives the best performance on a class of applications for a given fixed cost, k.
3. $V_T(T)$ the vector of values that gives the lowest cost for a given execution time, T, for a class of applications.

The efficiency optimized vector V_e describes a single point in machine space that represents the optimum machine size, grain size, and balance for a machine to provide throughput per unit cost. In a throughput oriented computing center, it would be better to have several V_e machines than fewer larger machines or more smaller machines. A common misconception is that V_e ought to be a uniprocessor. It seems intuitive to save on communication and spend all the money on processing. The flaw with this argument is that uniprocessors spend too much money on memory per processing element. To run a problem of a given size, two uniprocessors would have to have twice the memory of a single 2-processor parallel computer. The parallel computer would offer better cost-performance.

The cost vector, $V_k(k)$, defines a one dimensional locus of points in machine space. Each point on this contour describes the fastest single machine that can be built for some cost k. As k is increased, this vector is adjusted by investing in the element of the vector (P, p, m, c, b, l) that provides the best incremental return on investment. There exists some minimum cost, k_{min} for which a single-processor machine with a minimal processor and enough memory to hold the problem can be built. As k is increased from this point, the processor power is increased until the point is reached where it is more cost effective to add nodes than to add cost to a single node. This point defines the optimal grain size for a two-node machine. As the total cost, k, is increased from this point, the number of nodes, P, increases as does the optimal grain size. This increase in grain size with machine size occurs because larger machines require higher bandwidth communication networks and hence have a larger incremental cost of adding nodes.

As cost continues to increase, we eventually reach the cost of the optimal machine, k_e, where $V_k(k_e) = V_e$. Above this, machines continue to get more powerful, but with diminishing returns. As returns on investment continue to diminish, eventually, a point is reached where no further increase in performance can be realized. The lowest cost at which such performance can be realized defines k_{max}.

The performance vector, $V_T(T)$, defines the same locus of points in machine space as the cost vector, V_k. Consider machines as defining points in a cost-time (k,T) space. In this space, V_T and V_k define exactly the same set of points – with the axes interchanged. There is some minimum time, T_{min}, for which a machine built in a given technology can solve the problem at any price. As we increase T from this point, efficiency increases until we reach T_e where $V_T(T_e) = V_e$. Beyond this point, efficiency decreases until we reach, T_{max}, corresponding to the machine with cost k_{min}.

4 Cost Model

Now that we have qualitatively examined the tradeoffs involved with balance and grain size, we will derive an example cost model that will be used in the remainder of the paper to illustrate these tradeoffs quantitatively. The cost model derived here is based on CMOS microprocessors, commodity DRAM memories, and direct interconnection network technology. The first two are technologies that are used in almost all cost competitive computer systems today. Direct networks are widely used by many types of parallel computers. Many approximations are made to keep the cost model simple.

Other cost models are possible given a different base technology or a different set of approximations. While this may change the exact numerical results derived later, the methodology for determining balance and grain size remains the same. Also, as described above, the qualitative relationship between machine size, grain size, balance, and performance remains the same as long as the cost model is monotonic.

$K_n(p, m, c)$ gives the cost of a node as a function of the node configuration, (p, m, c). To first approximation, the total cost of the machine is,

$$K(V) = PK_n(p, m, c) \tag{1}$$

Later, we will more accurately account for global bandwidth and latency. We will consider the three components of the node (processor, memory, and communication interface) separately and compute the node cost as the sum of these component costs,

$$K_n(p, m, c) = K_p(p) + K_m(m) + K_c(c). \tag{2}$$

We use silicon area as a measure of cost for each component. Silicon area reflects the fundamental cost of building a component and thus is a good basis for comparing alternatives as opposed to market price which includes many artificial factors. To simplify calculation, we normalize cost to units of DRAM bits, viz. one bit of DRAM takes one unit of area and one unit of cost. We express the cost of other chips in terms of *DRAM bit equivalents* (Dbe). We assume that the chips from which a node is built are small enough that cost is approximately a linear function of area (i.e. we ignore the exponential cost factor due to low yield of large chips). Logic processes are often lower volume and hence higher cost than memory processes. We use a factor, k_l, the cost of a unit of logic area, to account for this difference.

4.1 Memory Cost

We approximate memory cost as a linear function of capacity,

$$K_m(m) = K_{ms}m + B_m. \tag{3}$$

Here, m is the memory size in words, K_{ms} is the cost per word of memory, and B_m is the fixed overhead cost of the memory. This overhead includes logic for translation, address decode, data multiplexing, and memory peripheral circuitry. For our calculations, we assume that $K_{ms} = W(wordsize) = 64$, and the overhead, B_m, is 10^5. This model ignores the cost of providing memory bandwidth for the sake of simplicity. Bandwidth is accounted for in calculating communication cost below.

4.2 Processor Cost

We model the relationship between processor cost, K_p, and performance, p, as an exponential curve reflecting a base cost with diminishing returns as cost is increased:

$$p = \begin{cases} 0 & \text{if } K_p < B_p \\ p_s(1 - e^{-(K_p - B_p)/K_{ps}}) & \text{otherwise} \end{cases} \tag{4}$$

A cost of B_p is required to achieve a minimal functional processor, perhaps a bit-serial integer unit with a few registers and no cache. As cost is increased beyond B_p, performance increases linearly at first with slope p_s/K_{ps}. This reflects the performance improvement gained by widening the data path, adding pipelined function units, dedicated floating-point hardware, and a modest-sized cache. As hardware is added, however, there are diminishing performance returns and performance saturates at an asymptote of p_s.

To model current logic versus memory costs, we set $k_p = 10$ Dbe. Studying the layout of some simple RISC processors [1, 7, 6] leads to a base cost of $B_p = 10^4 k_l = 10^5$ Dbe. That is, a minimal processor can be built in the area of 10K DRAM bits at a cost of 100K DRAM bits. A cost constant of $K_{ps} = 10^6 k_l = 10^7$ Dbe, and a p_s of one operation per cycle were arrived at from the study of some high-end processors [15, 13, 14].

Inverting (4) gives processor cost in terms of performance:

$$K_p(p) = B_p + K_{ps} \ln\left(\frac{p_s}{p_s - p}\right). \tag{5}$$

4.3 Communication Cost

Most routers for direct networks are I/O-bound chips. Thus, we model communications cost of our node as the area of a pin-bounded router chip:

$$K_c(c) = K_{cs}c^2 + B_c. \tag{6}$$

Since chip area grows as the square of the pads on the chip, cost is proportional to c^2, where c is the communication bandwidth in words/cycle. The communication cost factor, K_{cs} is the cost in DRAM bit equivalents of one word per cycle of I/O bandwidth. For our calculations, we use $K_{cs} = 4 \times 10^6$. We arrive at this by observing that $100c^2$ DRAM bits of area are required to provide perimeter space for c pads, W=64 pads are required for each word/cycle of communication bandwidth, and this area is $k_l = 10$ times more expensive than DRAM area. The base area for a router, B_c is estimated at 10^5 from a study of simple routers [3, 2, 1, 8].

5 Performance Model

To predict the performance of an application on a machine with a particular configuration, V, we characterize the application by its requirements vector, $R(N, P)$. Again, we divide the requirements into processor, memory, and communication components, $R_p(N, P)$, $R_m(N, P)$, and $R_c(N, P)$ respectively. Run an application of a problem size N on P processing nodes requires $R_p(N,P)$ processing operations per node, $R_m(N,P)$ words of memory per node, and $R_c(N,P)$ words of communication per node.

To simplify our calculation of performance, we assume that the resource demands are uniform over time and that processing and communication can be completely overlapped. Applications that are nonuniform, for example an application with several phases each of which has different requirements, can be handled by dividing the application into its phases, calculating the requirements for each separately, and applying our methods for ensembles of applications described in Section 7.3. Our assumption that communication and processing are overlapped imposes constraints on how the problem is structured and on the node architecture. In the node, this condition implies that the processor implements mechanisms such as prefetching or context switching to tolerate the latency of communications.

Given a requirements vector, R, and a machine configuration vector, V, we compute the performance (execution time) of an application as the maximum of its compute time and its communication time provided that there is sufficient memory to hold the problem:

$$T(R,V) = \begin{cases} \infty & \text{if } m < R_m \\ \max\left(\frac{R_p}{p}, \frac{R_c}{c}\right) & \text{if } m \geq R_m \end{cases} \tag{7}$$

The required number of operations, R_p, divided by the processing speed, p, yields the compute time. Similarly, the required number of words to be communicated, R_c, divided by the communication rate, c, gives the communication time. Our assumption that the processing and the communication can be completely overlapped allows us to use the max operator to obtain the effective run time.

6 Global Bandwidth and Network Latency

We now extend our cost and performance models to consider two additional properties of the network: global bandwidth and latency that capture the effect of communication locality on cost and performance. Per-node global bandwidth, b, is the bisection bandwidth of the machine divided by the number of nodes, P. Per-node latency, l, is the elapsed time required to complete a zero-length communication action across the space occupied by a single node. For regular mesh and torus networks or for networks where switch delay dominates wire delay, l corresponds to the time for a single hop. The portion of communication time due to message length is already accounted for by our bandwidth parameters c and b and thus is not included in l.

6.1 Cost Model Extension

To provide b words/cycle of global bandwidth per node on a machine packed in three physical dimensions, bP words/cycle must pass through the bisection plane of the machine. Since there are $(P^{\frac{1}{3}})^2$ nodes in the plane, there are $bP/P^{\frac{2}{3}} = bP^{\frac{1}{3}}$ words/cycle passing through the bisection area of each processing node. A node bisection area of $bP^{\frac{1}{3}}$ implies a node volume of $(bP^{\frac{1}{3}})^{\frac{3}{2}} = b^{3/2}P^{1/2}$. We model cost as being proportional to volume. Generalizing to n dimensions:

$$K_b(b) = K_{bs}b^{n/(n-1)}P^{1/(n-1)} + B_b. \tag{8}$$

Here K_{bs} is the cost of providing one word/cycle of bandwidth through the volume of a processing node. The base cost of global bandwidth is given by B_b. For our calculations, we use $K_{bs} = 10^6$ and $B_b = 10^5$.

Latency impacts the cost of both local and global communication. However, to simplify our analysis we consider the cost of latency separately from the cost of bandwidth. The minimum latency across a node is limited by physics to be l_{min}, the time of flight across a single node. We model cost by the following equation that approaches infinity as latency approaches this minimum:

$$K_l(l) = \frac{K_{ls}}{l - l_{min}} + B_l. \tag{9}$$

For our calculations we use $K_{ls} = 10^5$, $l_{min} = 0.1$, and $B_l = 0$. The base cost is set to zero as the base communications cost is already accounted for in B_c and B_b. The l_{min} of 0.1 reflects the ratio of switch delay to wire delay in current interconnection networks.

6.2 Performance Model Extension

To estimate the effect of global bandwidth on performance, we extend the requirements vector for an application, $R(N, P)$, with a global bandwidth component, $R_b(N, P)$, that gives the number of words of global communication per node required by the application. The required global communication, R_b, is that subset of the total communication, R_c, that is non-local and hence makes use of the network bisection. The execution time bound due to global communication is $R_b(N, P)/b$. For simplicity we consider an all-or-nothing model of local versus global communication. A more detailed model could use a hierarchical model of locality and characterize the machine by hierarchical bisection bandwidths. However, such a model would greatly complicate calculations with minimal effect on the final results.

Latency affects performance by introducing idle time during which the machine must wait for a communication operation to complete. We characterize the latency requirements of an application by R_l which denotes the total length (in units of the linear dimension of a node) of the communication operations along the critical path of the application. The minimum execution time of the program due to latency is then $R_l l$. If this is the largest term in our execution time equation, the computation is latency bound and this term gives the execution time. If there is a larger term, the computation is bandwidth bound and latency does not effect execution time.

R_l is a function of both the application and of the network on which it is run. For example, R_l for a switch-delay dominated multistage network is $log_2(P)$ times the number of messages in the longest path of the computation. For direct networks, R_l is the sum of the messages in the critical path of the computation, weighted by the number of hops traversed by each message.

To account for global bandwidth and latency we modify the execution time equation (7) as follows:

$$T(R, V) = \begin{cases} \infty & \text{if } m < R_m \\ \max\left(\frac{R_p}{p}, \frac{R_c}{c}, \frac{R_b}{b}, R_l l\right) & \text{if } m \geq R_m \end{cases} \tag{10}$$

7 Optimization

Now that we have derived formulae for cost, $K(V)$, and performance, $T(R, V)$, we can use them to constrain a search through machine space for an optimized machine. We allow cost to vary, and at each fixed cost point, we find the machine that maximizes performance. The resulting locus in machine space traced by this search is the vector, $V_k(k)$. This section discusses the details of this optimization procedure for individual applications and for an ensemble of applications.

7.1 Optimization in the Basic Model

Consider the basic model in the context of a single application. We make the observation that at all cost points, the optimal machine configuration always satisfies the following equations:

$$\frac{R_p}{p} = \frac{R_c}{c}, \qquad m = R_m \tag{11}$$

These equations are a statement of *balance*. The first equation states that communication and computation times should be equal. If they are not equal, we can take resources from the faster component without increasing runtime. The second equation states that the memory should exactly fit the problem. If the memory is larger than this amount, it can be reduced without impacting performance. When the processing and communication times are equal, and the memory fits the problem, the machine configuration, V, is balanced for requirements, R. In a balanced machine, each resource is utilized to its fullest. Under a cost constraint, the optimal machine will lie along the locus of points representing balanced machines; if a machine is not balanced, then the amount of underutilized resources can be reduced, decreasing cost. The balance constraint greatly reduces the size of the search space, and thus the complexity of the optimization procedure.

To illustrate our methodology, we now focus on optimizing a machine under a cost constraint for the two-dimensional Jacobi relaxation problem. Let the problem size be N and our cost point be k. We will calculate $V_k(k)$.

For block partitioned Jacobi relaxation, the application is characterized by the following functions:

$$R_p(P, N) = 4 + 4\frac{N}{P}, \quad R_c(P, N) = 8\sqrt{\frac{N}{P}}, \quad R_m(P, N) = 4 + \frac{N}{P} \tag{12}$$

Our balance constraints are:

$$m = R_m(P, N) = 4 + \frac{N}{P}, \quad \frac{p}{c} = \frac{4N/P}{8\sqrt{N/P}} = \frac{1}{2}\sqrt{\frac{N}{P}} \tag{13}$$

Our cost constraint is:

$$k = P\left(B_m + B_c + B_p + K_{ms}m + K_{ps}\log\left(\frac{p_s}{p_s - p}\right) + K_{cs}c^2\right) \tag{14}$$

Assuming a balanced configuration, our runtime is given by:

$$T(P, N, V) = \frac{\left(4 + \frac{4N}{P}\right)}{p} \tag{15}$$

The optimization process attempts to minimize T specified in Equation 15, subject to the constraints in Equations 13 and 14. The resulting vector V represents the optimal, balanced, machine for Jacobi. Section 8.1 presents and discusses the results for this optimization.

7.2 Optimization in the Extended Model

The analysis for the extended model is similar to the analysis for the basic model with the added terms for global bandwidth and latency. The balance constraint for optimized machines still holds, and the balance equations become:

$$\frac{R_p}{p} = \frac{R_c}{c} = \frac{R_b}{b} = R_l l, \qquad m = R_m \tag{16}$$

Consider the block partitioned Jacobi relaxation application in the context of the extended model. The application requirements in Equation 12 are still valid, but they need to be augmented by global bandwidth and latency requirements:

$$R_b(P, N) = 2\frac{\sqrt{N}}{P}, \quad R_l(P, N) = 1 \tag{17}$$

Observe that the latency parameter is one because the application displays perfect locality (see Appendix B for applications with more interesting global requirements).

Our balance constraints are:

$$m = R_m(P, N) = 4 + \frac{N}{P}, \quad \frac{4 + 4\frac{N}{P}}{p} = \frac{8\sqrt{\frac{N}{P}}}{c} = \frac{2\frac{\sqrt{N}}{P}}{b} = l \tag{18}$$

Our cost constraint assuming two dimensional networks is:

$$k = P\left(B_m + m + B_c + K_{cs}c^2 + B_p + K_{ps}\log\left(\frac{p_s}{p_s - p}\right) + B_b + K_{bs}b^2P + B_l + \frac{K_{ls}}{l - l_{min}}\right) \tag{19}$$

Assuming a balanced configuration, our runtime is the same as before, given in Equation 15. Section 8.1 presents and discusses the results from the extended model.

7.3 Optimization for Ensembles

To optimize a machine for a collection of applications or for a non-uniform application, we use ensembles. The ensemble is modeled as a set of requirements vectors; $R^i(N, P)$ denotes the requirement vector for element i of the ensemble. The optimization method depends on what kind of machine resource management is assumed, *space-sharing* or *time-sharing*.

In the space-sharing model, the machine is divided into partitions, allocating P_i processors for ensemble element i ($\sum_i P_i = P$). This models the case where a number of applications are running simultaneously on a machine as well as the case where a single application forks several processes that run in parallel on distinct nodes before joining. We assume that each node in the machine has the same values for p, m, and c. The execution time for a space-shared ensemble is the maximum execution time over its partitions. Assuming N_i is the size of the problem for application i,

$$T_{ss} = \max_i \left[\max \left(\frac{R_p^i(N_i, P_i)}{p}, \frac{R_c^i(N_i, P_i)}{c} \right) \right] \tag{20}$$

In our time-shared model, each application is run in sequence using the entire machine. This models the case where applications time-share the machine as well as the case of a single application that proceeds serially through a number of phases each with different requirements. In this case, execution time is the sum of the individual execution times:

$$T_{ts} = \sum_i \left[\max \left(\frac{R_p^i(N_i, P)}{p}, \frac{R_c^i(N_i, P)}{c} \right) \right] \tag{21}$$

In both the time-shared and the space-shared models, the memory constraint must be satisfied for each application by requiring that $m \geq \max_i(m_i)$.

The analysis for the space-sharing model is harder than that for the time-sharing model. In space-sharing, for a given machine size P, it is necessary to consider all possible partitions of those P processors into i partitions. In time-sharing, the search over these permutations is avoided since all ensemble applications run on the entire machine. For simplicity, we choose to only consider the time-sharing model when we present results for ensembles in Section 8.3.

Finally, Equations 20 and 21 are in the context of the basic model. Extending these to consider global bandwidth and latency is accomplished by inserting the terms for global bandwidth and latency:

$$T_{ss} = \max_i \left[\max \left(\frac{R_p^i(N_i, P_i)}{p}, \frac{R_c^i(N_i, P_i)}{c}, \frac{R_b^i(N_i, P_i)}{b}, R_l^i(N_i, P_i) l \right) \right] \tag{22}$$

$$T_{ts} = \sum_i \left[\max \left(\frac{R_p^i(N_i, P)}{p}, \frac{R_c^i(N_i, P)}{c} \frac{R_b^i(N_i, P_i)}{b}, R_l^i(N_i, P_i) l \right) \right] \tag{23}$$

8 Model Results

In this section, we study 4 applications in the context of our model: Jacobi, FFT, Nbody, and Matrix Multiply. We choose these applications because they are diverse and require conflicting machine requirements to run efficiently. The application requirements needed by the model for Jacobi were presented in Section 7; the application requirements for FFT, Nbody, and Matrix Multiply can be found in Appendix B.

8.1 Results for Individual Applications

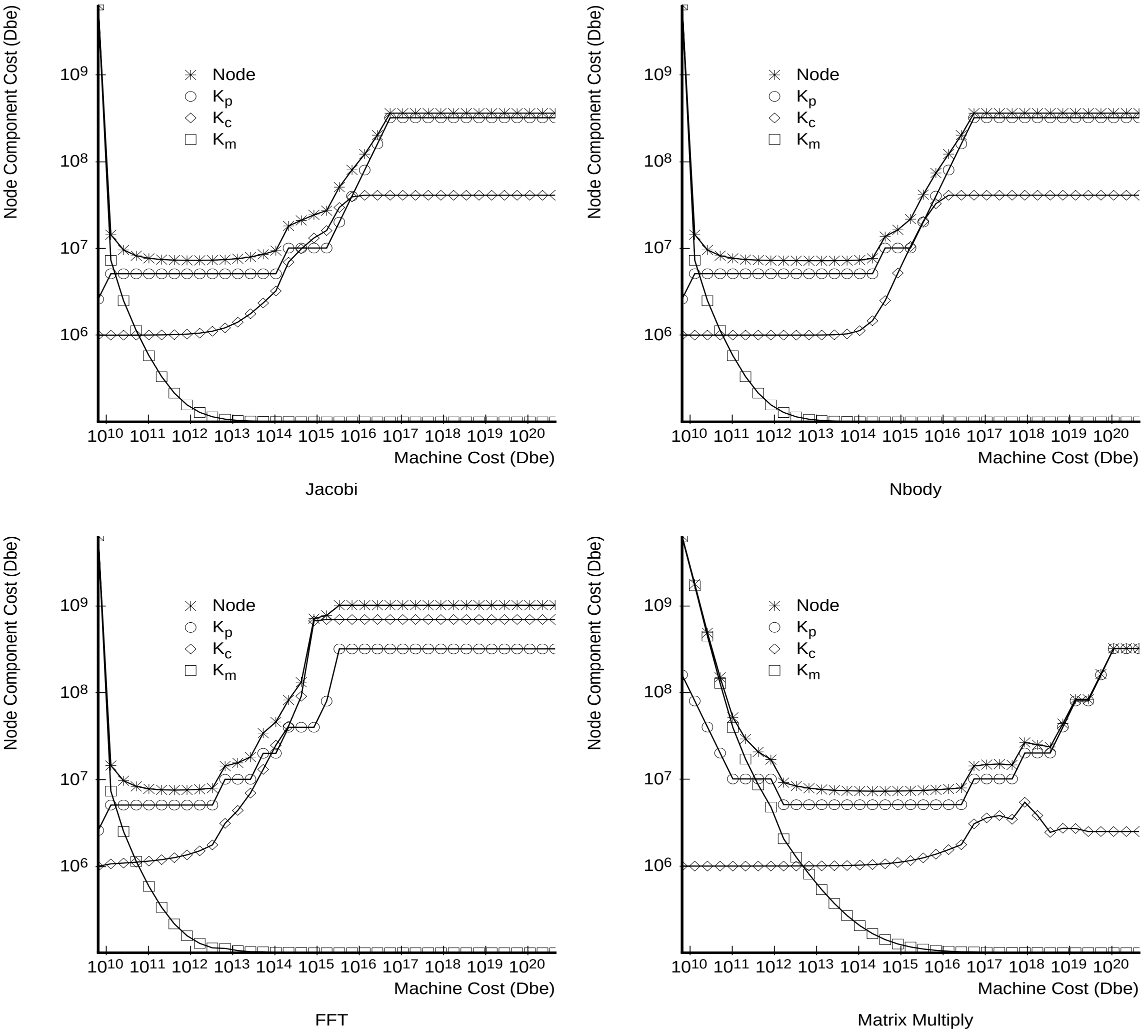
We first present results for the basic model when applications are considered individually. The analysis in this section assumes fixed problem size. The problem size for the Jacobi application is chosen to be 10^8 words; the problem sizes for the other applications are chosen such that all the applications require equal amounts of memory when executed on a single processor.

Figure 2 shows the breakdown of per-node cost into processor, memory, and communications components as a function of total machine cost, K. Figure 3 shows the number of processors and grain size as a function of machine cost. All costs are expressed in terms of DRAM bit equivalents (Dbe). Assuming DRAM cost of \$25/Mbyte, the range of machine cost spans \$30,000 to \$300 trillion. While this upper limit is ludicrous, we explore this region to show asymptotic behavior of the applications. In practice, considering machines above a cost of 10^{15} Dbe (about \$1 billion) is unrealistic.

The basic model predicts similar behavior for all four applications even though their requirements are diverse. Figure 2 shows per-node processor cost begins at some small value and increases to a plateau. This is true for all the applications except for Matrix Multiply whose processor investment starts at a higher value in order to yield a balanced solution. In the plateau region, greater incremental performance is gained by adding processors than by increasing the cost of a single processor; therefore, per-node processor cost remains fixed and machine size grows. As the number of nodes continues to increase, communication requirements to support these nodes increases as well. Eventually, communication cost is high enough to make the addition of more processors less attractive than increasing processor performance. This marks the end of the plateau region as processor cost again climbs, this time alongside communication cost which continues to increase in order to support the growing machine size. Finally, all machine parameters flatten out when application performance saturates. This happens when there is one processor for every element, and processor throughput becomes asymptotically close to 1 op/cycle. In theory, processor investment should continue to increase, but the diminishing return in processor performance is eventually undetectable by the precision of our calculations. The point of performance saturation occurs at a cost of k_{max}, and the execution time is T_{min}

The left graph in Figure 3 shows the trend of increasing machine size with increasing cost. Jacobi and Nbody reach the same limit of one processor per element as does FFT, but since FFT starts with a smaller problem size, its limit is actually lower. In Matrix Multiply, it is possible to have more processors than elements because the blocking creates more elements. The right graph of Figure 3 shows how computer grain size, or overall node cost decreases as the fixed amount of memory is divided over an increasing number of nodes; eventually, grain size levels off. In some applications, grain size actually increases slightly again when the optimal processor cost increases due to higher expense in communication to support a larger machine.

Figures 4 and 5 report results for the model when global bandwidth and latency considerations are included, and are analogous to Figures 2 and 3 for the basic model case. Similar trends that are observed in Figure 2 can be observed in the processor, communications, and memory components in Figure 4. This is especially true in the Jacobi and Nbody applications since both of these applications exhibit good locality and thus do not require great investments in global bandwidth and latency. FFT and Matrix Multiply, however, are harder to run on large numbers of processors. FFT inherently has poor locality, and blocked Matrix Multiply has poor locality when each processor has a small problem size (which occurs at large numbers of processors). At high machine budgets and large machine sizes, the performance of FFT and Matrix Multiply becomes latency-bound. This is reflected in a sharp increase in global latency investment, and



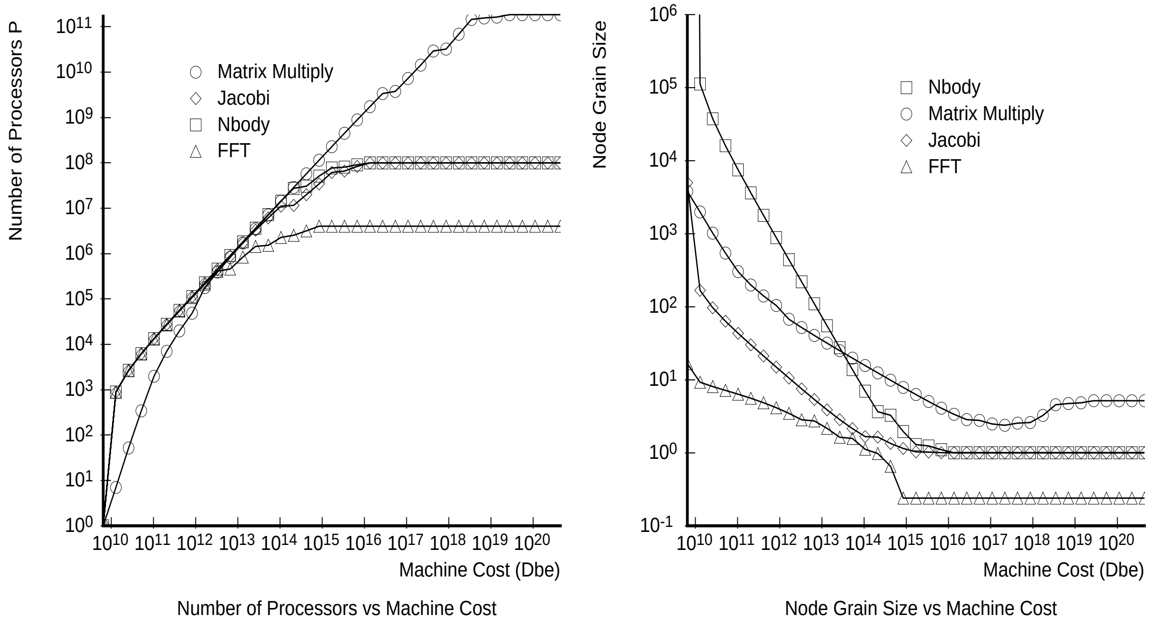
Jacobi

Nbody

FFT

Matrix Multiply

Figure 2: Node component costs for four applications as a function of machine cost under the basic model. K_p, K_c, and K_m are the per-node investments in processor, communications, and memory, respectively. The curve labeled "Node" shows the total per-node cost. All costs are in DRAM bit equivalents (Dbe).



Number of Processors vs Machine Cost

Node Grain Size vs Machine Cost

Figure 3: Number of processors and grain size versus machine cost under the basic model.

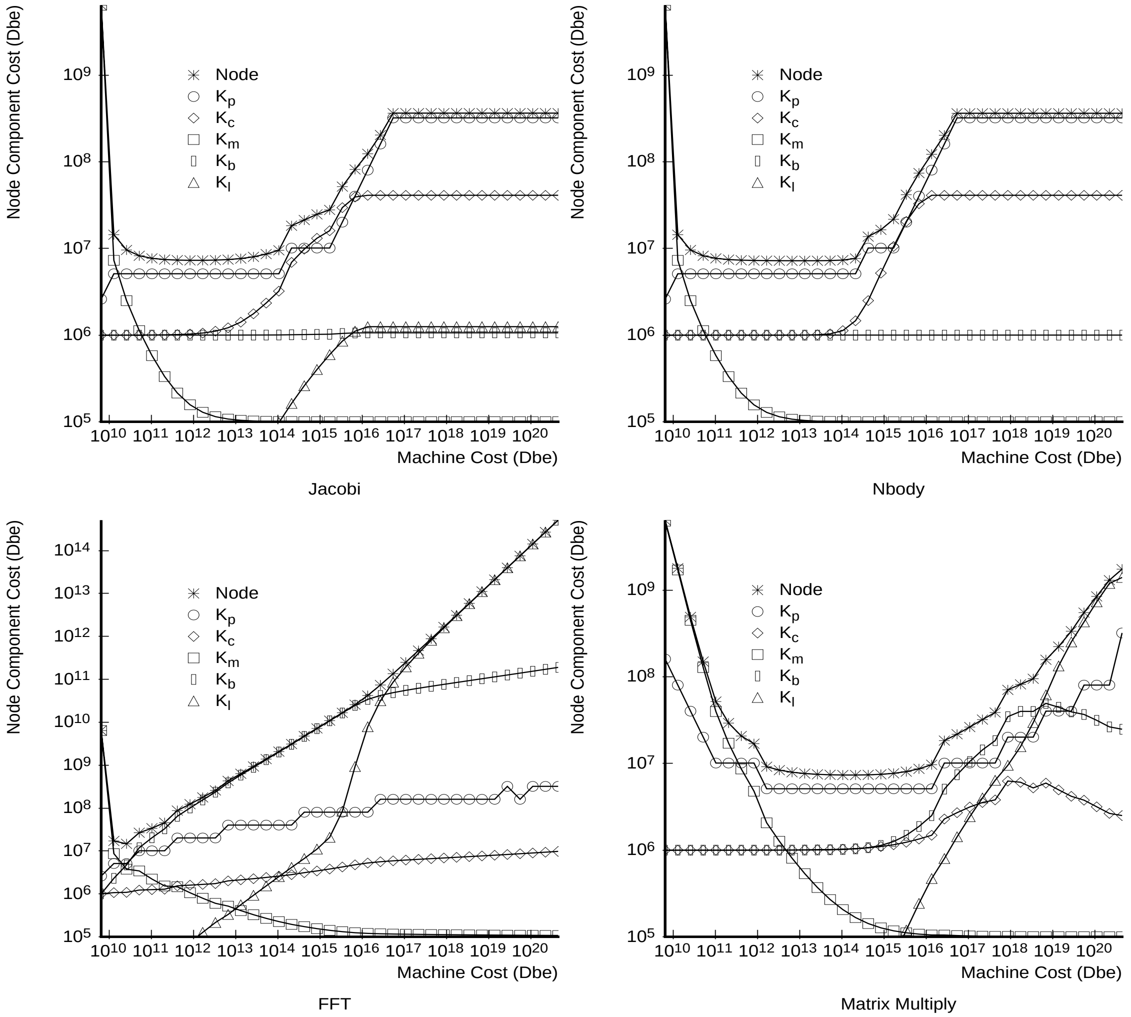
a much slower increase in machine size as can be seen in Figure 5.

8.2 Problem Scaling

In the previous analysis, problem size remains constant with respect to cost. This assumes the case in which more money is applied to run a given problem faster. Often, the motivation for building larger machines is to run problems that couldn't have been run with smaller machines, and this assumes scaling problem size with machine size. Under problem scaling, the curves presented in the previous section will change. One of the main differences is that at low-budgets, problem sizes will be smaller so that less of the total machine budget will be spent on memory. In the previous analysis, the cost of low-budget machines is dominated by the cost of memory. Also, at high budgets, fixed problem scaling results in performance saturation as the number of processors approaches the number of elements in the problem size. With problem scaling, larger problem sizes are chosen for large machines so that there is enough parallelism to support many more processors.

8.3 Results for Ensembles

The left and right graphs in Figure 6 show the results for a time-sharing ensemble under the basic model and extended model, respectively. These graphs show how well the applications run on a machine that is optimized for the ensemble of applications. Along the horizontal axis is the machine budget. The individually plotted points show slowdown of each application running on the ensemble machine compared to how it would run on a machine that is optimized for that particular application. The points with the line drawn through them show the performance of the entire ensemble of applications. This is the slowdown when all the applications are run on the ensemble machine in a time-shared manner compared against the sum of runtimes that would result if each application were run on a machine optimized for that application.



Jacobi

Nbody

FFT

Matrix Multiply

Figure 4: Node component costs for four applications as a function of machine cost under the extended model. K_p, K_c, K_m, K_b, and K_l are the per-node investments in processor, communications, memory, global bandwidth, and global latency, respectively. The curve labeled "Node" shows the total per-node cost. All costs are in DRAM bit equivalents (Dbe).

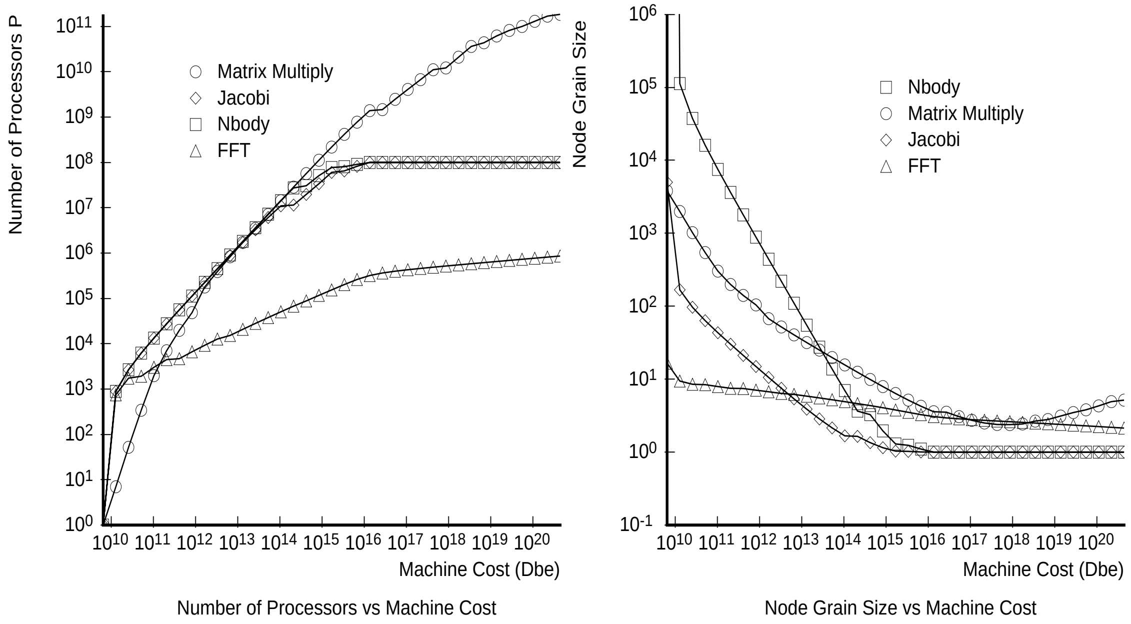


Figure 5: Number of processors and grain size versus machine cost under the extended model.

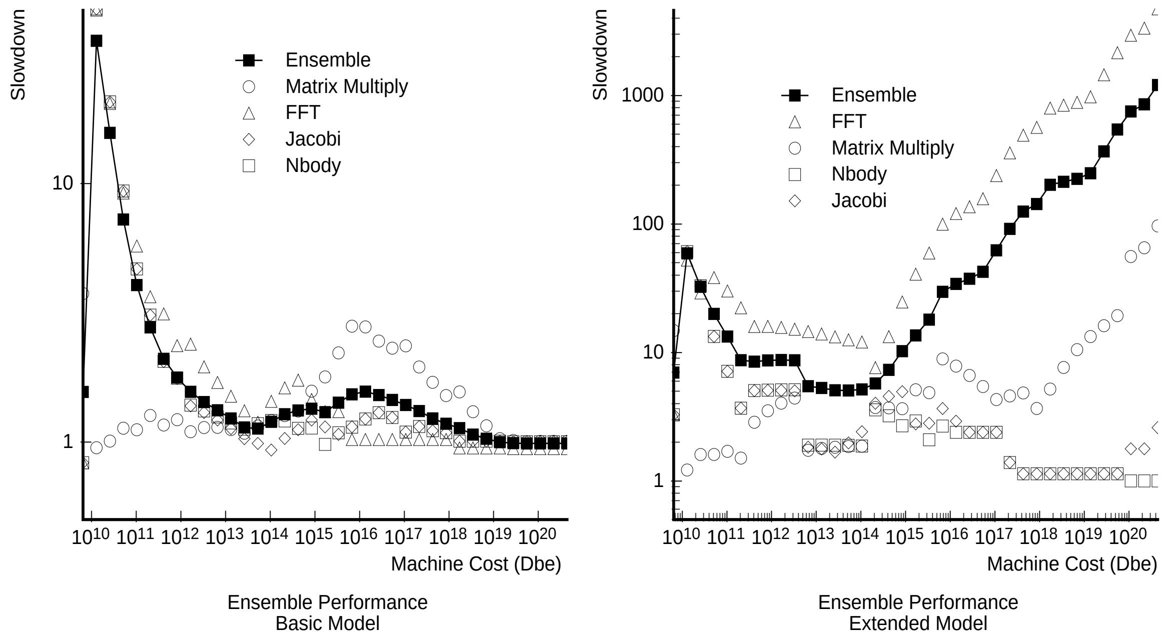


Figure 6: Ensemble performance under both the basic model and extended model.

In the basic model, there are three interesting regions of behavior. First, at low cost, there is not enough money in the budget to satisfy all the applications. Jacobi, Nbody, and FFT perform best when machine size grows quickly, but this is prevented by the memory requirements of Matrix Multiply. The total memory required to run Matrix Multiply grows with machine size, and at low budgets, there isn't enough money to support this memory cost for machine sizes that would be optimal for the other applications. Hence, Jacobi, Nbody, and FFT perform poorly and bring down the performance of the entire ensemble. Second, when the budget becomes large enough to afford the extra memory needed by Matrix Multiply, then Jacobi, Nbody, and FFT perform well, and the performance of the ensemble improves substantially. In this region, the performance of Matrix Multiply worsens slightly though. This is due to the fact that at higher budgets, Matrix Multiply is able to employ more processors than data elements, but the other applications cannot. In this case, the needs of the majority drive the design of the machine at the expense of Matrix Multiply performance. Finally, at extremely high budgets, enough money exists to satisfy the needs of all the applications reflected by the fact that all applications and the ensemble converge to a slowdown of 1, in the limit.

In the extended model, when global bandwidth and latency are considered, the results are similar to those in the basic model. The design of the ensemble machine is initially driven by Matrix Multiply because of its memory requirements making ensemble performance poor, and when these memory needs can be fulfilled by increasing budget, ensemble performance improves. The behavior of the extended model departs from that of the basic model at large budgets where slowdown begins to increase and continues to worsen with increasing budget. At high budgets, Jacobi, Nbody, and Matrix Multiply prefer many processors. FFT, being an application with poor locality, requires very high cost in global latency to perform well at these large machine sizes. Even at our ludicrously high budgets, there is not enough money to satisfy the global latency requirements for FFT. Had we extended our cost range even further, eventually, the poor performance would reach a maximum, turn around, and improve to 1 in the limit.

8.4 Looking at Real Machines

The results thus far have examined the predictions of the model for ideal machines. We would now like to compare these predictions against existing machines. In particular, we would like to investigate the per-node memory investments of current machines and see how they compare to model predictions. We do this for two experimental machines, the Alewife [9] machine, and the J-Machine [10], and two production machines, the CM-5 [11], and the Cray T3D [12].

Figure 7 plots per-node memory cost as a function of number of processors, P. The solid lines show the per-node memory cost of the ideal machine as predicted by the model in the ensembles analysis. There are five such curves, each corresponding to a different problem size (these are the problem sizes of the Jacobi application in the ensemble). The dashed lines show the per-node memory costs of the 4 machines which we consider. These curves are horizontal since per-node memory does not change with machine size. For each machine (dashed line), we can find the points of intersection with problem sizes (solid lines), and at these points of intersection, read off the ideal machine size (x-axis). For a given problem size, this number is the most cost-effective machine size as predicted by the model. For instance, with a problem size of 10 million elements, we see that the Cray is cost-effective at about 10 processors, the CM-5 at low 10s of processors, the Alewife machine at low 100s of processors, and the J-Machine at low 1000s of processors. The production machines aren't cost-effective in massively parallel configurations unless the problem size is astronomical in size.

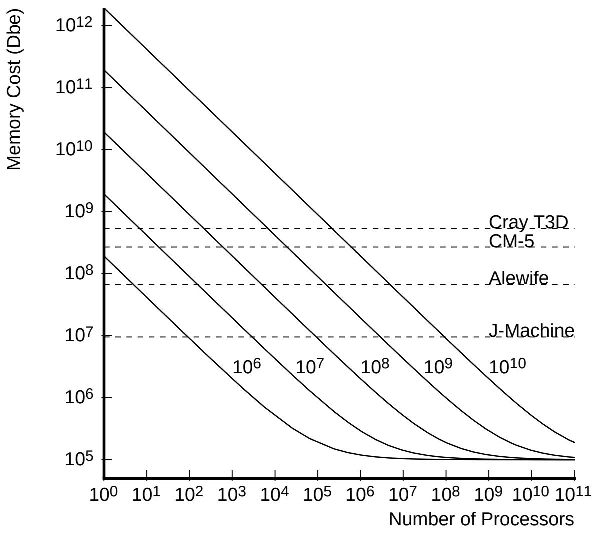


Figure 7: Per-node memory cost versus number of processors for the ensemble analysis over five different problem sizes from 10^6–10^{10}. Dashed lines show per-node memory cost for four existing machines.

9 Conclusion

This paper provides a framework upon which engineers can reason about the design of parallel processors. The framework uses a "blc mpP" machine characterization that considers processing, memory, local and global communication, and latency as separate machine resources. This is a unique characterization of machine space since it captures the effects of locality by treating local and global communication separately. The framework recognizes the importance of balance in good design, and integrates this idea with a cost and performance model to provide a useful design tool. We feel this is an important tool because it allows design to be driven by analysis rather than by rules of thumb which often lead to unoptimized designs. Having provided this framework, this paper arbitrarily chooses a diverse application suite in order to exercise the model and to address some general questions in parallel computer design.

One question addressed by the results obtained from our model is the feasibility of a general-purpose parallel computer. Reassuringly, our model predicts that general-purpose machines are possible, at least for the application suite we chose. Evidence for this can be found in the ensemble results. There are cost regions in which it is possible to build a machine that is close to optimal for all the applications that we studied. Unfortunately, the costs for these machines are mostly above the cost range for existing parallel computers. In the cost regions where performance is poor, we saw two factors that degraded ensemble performance. At low-budgets, the needs of applications with large memory requirements take money away from building more processing nodes for less memory-bound applications. At extremely high budgets (although granted, we may never see machines this big), there is a conflict between applications with good locality, and those with poor locality. Applications that are scalable prefer large machine sizes. Applications that have unscalable communication patterns cannot afford the investment needed in reducing global latency to support large machines. When it is not possible to fulfill the needs of different applications given a budget, and the designer has some knowledge about which applications are most important, it may be wise to give up the hope of building a general-purpose machine and to drive the machine design in an application-specific manner.

The second question our model addresses pertains to node grain, and in particular, the amount of per-node memory in a cost-effective design. We find that the CM-5 and Cray T3D, for reasonable problem sizes, are cost-effective at relatively small machine sizes, 10s to low 100s of processors. If these node architectures were to be used in a moderately to massively parallel machine, far too much of the total machine budget would be spent on memory. These machines are, however, cost-effective at large machine sizes if they are used to run extremely large problem sizes.

A Comprehensive List of Notation

p	processing power per node (operations/cycle).
m	memory size per node (words).
c	communication bandwidth per node (words/cycle).
b	global communication bandwidth per node (words/cycle).
l	communication latency per node for zero-length global message (cycles).
P	number of nodes.
N	problem size
V	A machine configuration vector: P, p, m, c, b, l.
V_e	Machine configuration giving the highest efficiency (performance/cost).
$V_k(k)$	Machine configuration giving the highest performance on a set of problems for a given cost, k.
$V_T(T)$	Machine configuration giving the lowest cost on a set of problems for a given performance, T.
$K(V)$	Cost of a given machine configuration, V (in DRAM bit equivalents or Dbe).
$K_n(p, m, c)$	Cost of a node with configuration, (p, m, c) (Dbe).
$K_m(m)$	Cost of memory with capacity m (Dbe).
$K_p(p)$	Cost of a processor with performance p (Dbe).
$K_c(c)$	Cost of communications with bandwidth c (Dbe).
$K_b(b)$	Cost per node of global communication with bandwidth b (Dbe).
$K_l(l)$	Cost of supporting communication latency l per node (Dbe).
k_{min}	Minimal machine cost (Dbe).
k_p	$= k_c = k_b = k_l$ cost of a unit of logic area (Dbe).
K_{ps}	processor cost factor, cost of reaching $(1 - e)$ of saturation performance (Dbe).
K_{ms}	memory cost factor, cost of one word of memory (Dbe).
K_{cs}	communication cost factor, the area in DRAM bits of one word of I/O pads (Dbe).
K_{bs}	global bandwidth cost factor, cost of one word per cycle of global bandwidth (Dbe).
K_{ls}	latency cost factor, cost of one cycle of latency (Dbe).
B_m	base cost of memory (Dbe).
B_p	base size of processor (Dbe).
B_c	base size of local communications component (Dbe).
B_b	base size of global communications component (Dbe).
$R(N, P)$	requirements vector for an application.
$R_p(N, P)$	required number of processing operations per node.
$R_m(N, P)$	required amount of memory words per node.
$R_c(N, P)$	required number of words of local communication per node.
$R_b(N, P)$	required number of words of global communication per node.
$R_l(N, P, V)$	latency inherent to the computation.
W	Wordsize (bits)
T	The amount of time required to solve a set of problems (cycles).
T_e	The amount of time required by the optimal machine configuration, V_e, to solve a set of problems (cycles).
T_{min}	The minimal time to solve a set of problems by any machine configuration (cycles).
T_{max}	The time required by the minimal machine configuration to solve a set of problems (cycles).

B Input Parameters for Several Applications

The global bandwidth and latency calculations in the following examples assume that the network is a k-ary n cube.

B.1 Jacobi-2D

For Jacobi-2D, the problem related functions are:

$$R_p = 4 + 4N/P$$

$$R_c = 8\sqrt{\frac{N}{P}}$$

$$R_m = 4 + \frac{N}{P}$$

$$R_b = 2\frac{\sqrt{N}}{P}$$

$$R_l = 1$$

B.2 Blocked FFT

For blocked FFT, the problem related functions are:

$$R_p = 3\left(1 + \frac{N}{P}\right) log_2 N$$

$$R_c = 4\frac{N}{P} log_2 N / log_2(N/P)$$

$$R_m = \frac{N}{P} log_2 N$$

$$R_b = R_c = 4\frac{N}{P} log_2 N / log_2(N/P)$$

$$R_l = nP^{1/n} log_2 N / log_2(N/P)$$

B.3 N Body

For the N body problem, the problem related functions are:

$$R_p = 2\frac{N^2}{P}$$

$$R_c = 2\left(N - \frac{N}{P}\right)$$

$$R_m = 1 + \frac{N}{P}$$

$$R_b = \frac{N}{P}$$

$$R_l = nP^{1/n}$$

B.4 Blocked Matrix Multiply

For blocked matrix multiply, the problem related functions are:

$$R_p = max\left[2\frac{N^3}{P}, \quad 1 + log_2 N\right]$$

$$R_c = 3\frac{N^2}{P^{2/3}}$$

$$R_m = \frac{N^2}{P^{2/3}}$$

$$R_b = \frac{N^2}{P}$$

$$R_l = P^{1/6}$$

References

[1] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, and Wen-King Su. The Design of the Caltech Mosaic C Multicomputer. *Research on Integrated Systems, Proceedings of the 1993 Symposium*, The MIT Press. Cambridge, Massachusetts, 1993. pp. 1-22.

[2] Peter R. Nuth and William J. Dally. The J-Machine Network, *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1992. pp. 420-423.

[3] Charles L. Seitz and Wen-King Su. A Family of Routing and Communication Chips Based on the Mosaic. *Research on Integrated Systems, Proceedings of the 1993 Symposium*, The MIT Press. Cambridge, Massachusetts, 1993. pp 320-337.

[4] H. T. Kung. Memory Requirements for Balanced Computer Architectures, *IEEE* 1986. pp. 49-54.

[5] Thomas J. Holman and Lawrence Snyder. Architectural Tradeoffs in Parallel Computer Design. *Advanced Research in VLSI, Proceedings of the 1989 Decennial Caltech Conference*, The MIT Press. Cambridge, Massachusetts, March 1989. pp. 317-334.

[6] Paul Chow The MIPS-X RISC Microprocessor, *Kluwer Academic Publishers*, August 1989.

[7] William J. Dally Architecture of a Message-Driven Processor, *Proceedings of the 14th Annual Symposium on Computer Architecture*, June 1987, pp. 189-196.

[8] William J. Dally and Charles L. Seitz. The Torus Routing Chip, *Distributed Computing*, Volume 1. pp. 187-196.

[9] Anant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. *Proceedings of the Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. Also appears as MIT/LCS Memo TM-454, 1991.

[10] William J. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer, Proceedings of the IFIP (International Federation for Information Processing), 11th World Congress, *Elsevier Science Publishing*, New York, 1989. pp. 1147-1153.

[11] CM5 Technical Summary, Thinking Machines Corporation, Cambridge, MA. Oct, 1991.

[12] CRAY T3D System Architecture Overview, Cray Research, Inc. Revision 1.C, September 23, 1993.

[13] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 Superscalar RISC Microprocessor, *IEEE Micro*, Volume 2, Number 2, April 1992. pp. 40-63.

[14] Dennis Allison and Michael Slater. National Unveils Superscalar RISC Processor, *Microprocessor Report*, Volume 5, Number 3, February 20, 1991.

[15] Daniel Dobberpuhl et. al. A 200 Mhz 64b Dual-Issue Microprocessor, *IEEE Solid State Circuits Conference*, Volume 35, February 1992. pp. 106-107.