Analysis of Task Mapping for Parallel Supercomputers

A Dissertation Presented

by

Janet Laura Braunstein

to

The Graduate School

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

in

Applied Mathematics and Statistics

Stony Brook University

August 2007

Abstract of the Dissertation

Analysis of Task Mapping for Parallel Supercomputers

by

Janet Laura Braunstein

Doctor of Philosophy

in

Applied Mathematics and Statistics

Stony Brook University

2007

This thesis concentrates on mapping applications to parallel computers with complex network architectures. The common practice of assigning tasks to processors without regard to the communication pattern of the problem or the network topology of the machine is an inefficient one. This approach has not caused any serious performance degradation for systems with small numbers of processors connected by simple, usually all-connected, networks. However, near-optimal performance for most general applications and architectures cannot be achieved without the incorporation of a sound mathematical model which represents the problem and the machine and predicts the relative runtimes for various mappings.

Many models have been developed, each appropriate under some circumstances. Few, however, deliver decent performance for selected scientific computing applications on a variety of architectures. We analyze the performance of several models on an assortment of problems on four computers with different network topologies. We attempt to improve upon models currently in use by developing methodologies to incorporate factors that are recognized as significant yet often ignored or poorly represented.

The two major problems studied in this thesis are integral components of many common applications: matrix multiplication and the fast Fourier transform. Each has been implemented on a Beowulf cluster, a distributed symmetric multiprocessing system, and two cellular architectures of differing topology. Our results reveal the great dependence of the performance of an application on the mapping model.

In addition to illustrating the significance of task mapping, we also address the difficulty that determining an efficient model can be a time-consuming operation. Our work seeks to remedy this problem by proposing guidelines for choosing an optimal method of task assignment, based on the applications and the architectures of the networks to be utilized. The goal is to use these guidelines as the foundation for a much more desirable programming paradigm: automatic parallelization.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Before a parallel program can be created, there are several important decisions to make. First, the corresponding sequential program must be decomposed into individual tasks that can be assigned to a network of processors. This is called task partitioning. Second, once the partitions are established, the tasks must be assigned to processors. This is referred to as task mapping. Finally, task scheduling, where applicable, decides the order in which the tasks should be executed. All three of these problems are related and are known to be NP-complete. The present thesis work focuses on task mapping.

Many researchers in the area of parallel computing turned their attention to task mapping throughout the decade ending in 1995, while the next several years saw a decline in study of the subject. Recently, interest has returned, and task mapping is now becoming a major component for achieving better efficiency in parallel computing. The goal is to design algorithms and the related software for task assignment for general problems on any massively parallel system with thousands or more processors. This challenging endeavor of automatic parallelization is a long-term goal; in the meantime, smaller projects have been undertaken. Some involve determining an efficient model for a specific application on a specific topology; others focus on a particular application on general topologies or general applications on a particular topology [11, 24, 28, 37, 46, 47, 55, 62]. Some emphasize the objective function [47] while others emphasize the heuristic [8–10, 15, 24, 28, 40, 49, 60, 65–67]. Some use real applications to test their ideas; others do not [9, 37, 47]. Some use simulators only rather than real parallel machines [22, 24, 55]. Some only look at theoretical objective function values for applications rather than experimental results [60]. A move toward automatic parallelization has been made by examining these specific instances as a group and setting forth strategies based on the patterns found and other observations made.

Both assignment and coding can be done manually or automatically. Figure 1 illustrates the four subsequent types of mapping-implementation combinations. The most primitive combination is manual mapping and manual implementation (MMMI), which involves examining the problem and the machine architecture and making an educated mapping decision without the aid of any tools. Subsequently, a parallel program is written by hand; the communication portion is manipulated to take advantage of the mapping. This method relies solely on human experience and does not solicit any type of automation. Optimality may not be achieved in many cases, particularly those involving complex applications or architectures. We choose not to examine manual mapping and automatic implementation (MMAI). Manual assignment is unscientific and often leads to poor performance; automatic programming relies more heavily on the tools of computer science than those of mathematics. In automatic mapping and automatic implementation (AMAI), the most advanced of the four, we can envision a database consisting of a large collection of possible models. Given network configuration, the application, and possibly the problem size (as done in [36]), the best model will be chosen and applied, producing efficient code. The focus of this thesis work is automatic mapping and manual implementation (AMMI). In this case, a model is used to determine the best mapping. Given the mapping, parallel code is written by hand.

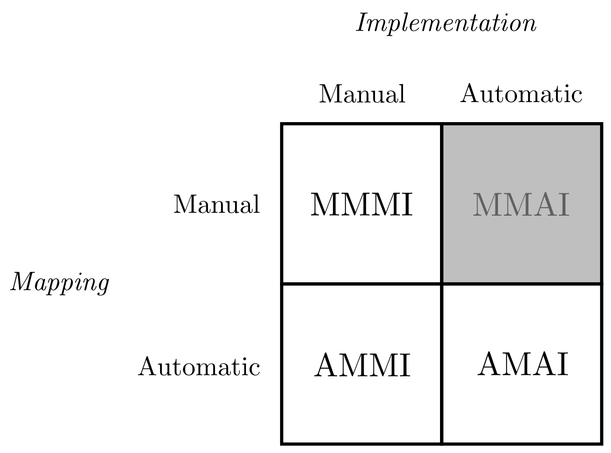

Figure 1.1: The four types of mapping-implementation combinations

The dissertation is organized as follows. The focus of Chapter 2 is on the definition of the task mapping problem and a related literature search. We describe the platforms on which our ideas are tested in Chapter 3. We introduce our new work and results obtained in Chapter 4. We discuss the future of task mapping in Chapter 5. Finally, in Chapter 6, we summarize the dissertation and draw general conclusions from the body of architectures and applications studied in Chapter 4.

Chapter 2

The Task Mapping Problem

The task mapping problem deals with the assignment of tasks to processors, subject to a set of constraints, in such a way that a given objective function, usually attempting to represent the time for completing an application on the given platform, is optimized. The tasks are the sequential sub-problems into which the application has been decomposed; some can be executed in parallel. The processors are the individual computing elements of a parallel system. The constraints may describe such conditions as maximum number of tasks to be mapped to a processor. The objective function quantifies the mapping and measures factors such as cost of inter-processor communication and computational load balance. It should be designed with an implementation strategy already in mind. Finding an efficient mapping leads to minimization of time to solution, or total execution time. Because the general task mapping problem is NP-complete, the standard approach is to employ heuristics to obtain solutions in a shorter time than total optimality would require.

Graphs can express information about applications and architectures that will be relevant to the parallelization process. An application can be represented by a task interaction graph (TIG). This is an undirected graph $G_D = (V_D, E_D)$ where the set of vertices V_D represents the tasks and the set of edges E_D represents the necessary communications among tasks. More specifically, each edge $(i, j) \in E_D$ represents necessary communication between tasks i and j at the completion of the computation step by the processors associated with tasks i and j. Each edge has an associated weight $w(i, j)$, denoting the amount of communication necessary. The TIG assumes that all tasks may be executed simultaneously and independently. Weights may also be associated with the vertices, denoting computation cost.

We can also describe the computer architecture itself by a processor communication graph (PCG). In this graph $G_S = (V_S, E_S)$, V_S is the set of processors, and the weight of each edge $(i, j) \in E_S$ denotes the cost of communication between processors i and j. Edges defined as such exist only between directly-connected processors.

The mapping problem translates into finding a function $f : V_D \rightarrow V_S$ which maps each task to a single processor such that a chosen objective function is optimized. This problem is known to be NP-complete.

For the purposes of this work, we view the graphs described above as matrices to express information about applications and architectures that will be relevant to the parallelization process. We define two matrices to aid in explaining the application and architecture, i.e., the supply matrix and demand matrix. We assume n number of tasks and p number of processors, and further assume that the tasks have been numbered $0, 1, 2, \ldots, n-1$ and the processors have been numbered $0, 1, 2, \ldots, p-1$.

Definition 2.0.1 *The supply matrix is a $p \times p$ structure in which each entry (i, j) represents the cost of communication from processor i to processor j.*

Definition 2.0.2 *The demand matrix is an $n \times n$ structure in which each entry (i, j) represents the amount of communication necessary from task i to task j.*

We may also refer to the supply matrix as the machine matrix because it is dependent on the properties of the computer architecture, particularly the network architecture. We may refer to the demand matrix as the application matrix because it characterizes the communication properties of the application and underlying solution algorithms.

A supply matrix can easily be obtained from a PCG by running an all-pairs shortest path algorithm. We lose some information in moving from graph to matrix representation. We no longer know how processors are connected; we simply know how far apart they are. We contend that the negative effect of this loss of information is negligible, however. With the point-to-point communication ability afforded by MPI, the programmer does not need to be concerned with the exact path a message will take. Storage requirements for a matrix are smaller than those for a graph, and algorithms may be run more quickly on matrices.

For a given computer, we can define at least two types of supply matrices, theoretical and experimental. The theoretical supply matrix is based solely on the description of the network architecture. The simple version of cost of communication can be defined as the number of hops between the processors. For machines with a 3D or 6D torus architecture like Blue Gene/L and QCDOC, described in Chapter 3, we may also want to study the effect of using a weighted form of the hop count, depending on the number of torus dimensions used in the shortest path from processor i to processor j, or using the Euclidean distance between processors (see equation 6.3)[11]. The cost of communication in the experimental supply matrix is defined to be the experimental time to send from one processor to another. The experimental matrix is obtained by running tests to determine the actual cost of sending data from one processor to another.

In the demand matrix, we define the amount of communication necessary as the number of message sends. We may refine the model by incorporating message size in future work. Although we create our demand matrices by hand because of our small number of tasks, we assume that these matrices can be easily created [11, 51] in any instance through the use of MPI tracing and profiling tools. These tools, such as Vampir, from Pallas, and Jumpshot, part of an MPICH distribution, return detailed information regarding the communication pattern of an application. For a given task, it can be determined which other tasks it communicates with, as well as the volume of data communicated.

The trend toward more complicated and irregular, for example, nonuniform memory access, supercomputer architectures has brought about growing recognition of task mapping as a powerful tool for increasing efficiency. In a cyclic fashion, advancements made in the field of task assignment may also lead to new network designs to further increase the speed of applications.

2.1 Mapping Models

We define the input to a task mapping problem to be the set of factors which are fixed before the execution of the task mapping procedure. The input consists mainly of the characteristics of the applications to be processed in parallel and those of the computer networks utilized. Inputs can vary in the following areas:

- upper-level architecture of system (homogeneous or heterogeneous)
- network characteristics, particularly the topology (linear array, tree, ring, mesh, hypercube, torus, etc.)
- communication patterns of application
- computation requirements of application

We define the model to be the chosen representation of the proposed solution method for the given input. It can be comprised of a number of components, including the following:

- representations (graphs or matrices)
- objective function (minimizing cost of communication and/or computation)
- desired quality of solution (optimal or quasi-optimal) or stopping condition
- optimization method (simulated annealing, genetic algorithm, etc.)

The input and model combine to form an instance of the task mapping problem.

Because the objective function quantitatively differentiates mappings from each other, it is the basis upon which assignments are ordered in terms of their efficiency. For this reason, it is the focal point of the model. If it is designed correctly, mappings that produce the same objective function value should be equally efficient. Time to solution is a function of computation and communication costs; we therefore seek to achieve both computational load balance and minimal inter-processor communication cost [15] when the objective function is optimized. Computation costs can be determined with a fairly high degree of accuracy because they depend mainly on the speed of each processor. Communication costs cannot be expressed as easily because they are affected by many runtime factors, outlined below.

The time taken to communicate a message of size n bytes from one location to another may be viewed as

$$t_c = l + \frac{n}{b}, \tag{2.1}$$

where l represents latency, in seconds, and b represents bandwidth, in bytes per second. Latency measures the speed of the network, or the time it takes for a packet to cross a network connection from sender to receiver. It is the time interval between the instant at which an instruction control unit initiates a call for data transmission and the instant at which the actual transfer of data begins. Latency is related to the hardware characteristics of the system and the layers of software that are involved in initiating the task of packing and transmitting the data. Bandwidth measures the capacity of the network, or the amount of data that can be transmitted across a network connection in a fixed amount of time. It is the data transfer rate. Neither latency nor bandwidth is constant for different pairs of messengers on a given architecture; rather, both are dependent on the relative locations of the sender and receiver, the availability of buffers, and contention for communication links [11]. The relative importance of latency and bandwidth varies with communication types. For fine-grained applications, which are characterized by many small messages, latency is more significant. In the case of course-grained algorithms involving fewer larger messages, bandwidth is more significant.

Some models choose an objective function that measures communication cost as the sum of all costs induced by the mapping. However, this does not take into consideration the fact that some communication can occur simultaneously, i.e., parallel message passing. In other cases, computation and communication costs are summed for each processor, and the objective is to minimize the maximal cost over all processors. Although this method is suitable for applications in which all processors should be in the same type of phase at any given time, it is inappropriate in cases where certain tasks must be completed before others are started. Network congestion can also affect communication time in ways that are not fully predictable. For some machines, the exact route a message will follow from one processor to another is not known prior to execution of the program. The paths that two messages plan to take may not be edge-disjoint. Because of the difficulty in accurately incorporating these and other factors into the model, some aspects of the actual messaging costs are often unaccounted for. Care must be taken to produce an objective function that reliably ensures mapping efficiency.

We examine existing models with the goal of selecting and applying aspects of several of them to our models. We attempt to improve upon them by making the theoretically best mapping determined by the objective function more consistent with the experimentally best mapping.

2.1.1 Communication-only models

Although total execution time generally reflects a combination of computation time and communication time, mapping models do not necessarily have to include both of these factors. When homogeneous collections of processors and applications that have been partitioned into tasks of equal computational weight are involved, removing computation time from the model may result in increased manageability with negligible effect on quality of solution obtained. Even cases that do not meet the criteria above but are highly communication-intensive may be good candidates for communication-only models.

Task mapping models have evolved greatly since their introduction. In his classic 1981 model, Bokhari [12] proposed the objective of maximizing the number of pairs of communicating tasks placed on pairs of directly-linked processors on the network. All tasks are assumed to be computationally equivalent, and there is to be a 1-to-1 mapping of tasks to processors. The cost of communication for all pairs of directly-connected processors is also assumed to be equal. When viewed from a graph perspective, Bokhari's objective corresponds to assigning TIG vertices to PCG vertices in such a way as to maximize the number of TIG edges whose endpoints map to PCG vertices connected by edges of minimal weight.

Bokhari's model begins with an initial random mapping. From there, an exhaustive hill-climbing heuristic is applied; a check of all possible pairwise interchanges of task-processor combinations is made, and the one that most favorably affects the cost function is selected. This process is repeated until the top of the hill (the mapping for which there are no beneficial interchanges) is reached. The cost of this mapping is recorded, and a random jump is made to a new mapping for which the process is repeated. At the top of each hill, a comparison is made to the best result obtained thus far, replacing it if better. Results were obtained for applications of up to 49 tasks.

A shortcoming of this model is that it creates a black-and-white distinction rather than a continuous one. The objective function gives a mapping one credit for each time it places communicating tasks on directly-connected processors. Such a function fails to distinguish between placing communicating tasks on processors two hops away and many more away. In both cases, zero credits are added. With today's growing trend toward massively parallel supercomputers, this is likely to be a costly oversight, ultimately resulting in the failure of the model.

Several subsequent models have attempted to build on Bokhari's idea but avoid the weakness of his objective function by adopting a quadratic assignment problem (QAP) formulation from the field of operations research. Given a set P of facilities and a set L of locations, the standard QAP attempts to find a mapping $f : P \rightarrow L$ that minimizes the objective function

$$\sum_{p_1,p_2 \in P} w(p_1,p_2) \cdot d(f(p_1), f(p_2)), \tag{2.2}$$

where $w(p_1, p_2)$ is the amount of supplies that must be transported between facilities p_1 and p_2, and $d(f(p_1), f(p_2))$ is the distance between the location to which facility p_1 is mapped and the location to which facility p_2 is mapped. The traveling salesman problem is a constrained instance of the QAP.

Three recent models [8, 11, 55] adopt this QAP objective function with the goal of minimizing latency and link contention and therefore total communication time. Link contention is not explicitly addressed in the objective function, but the belief is that the optimization process will produce a mapping that results in collision reduction. The objective function encourages small values for inter-processor distance for communicating tasks. This means that we can expect the optimization process to result in a mapping for which message path length in general is relatively short. In turn, there are fewer potential possibilities for two messages attempting to use the same communication link. The names given to the metrics are different for the three models, but all measure the same quantity.

In 2001, Moh et al. [55] built on Bokhari's idea of attempting to map communicating tasks to nearest-neighbor processors. Studying special cases of switch-based networks of irregular topology, they defined the quantity to be minimized as weighted cardinality, $|f_m|$, which is expressed as

$$\sum_{(x,y) \in E_v} F_v(x,y) \cdot F_p(f_m(x), f_m(y)), \tag{2.3}$$

where x and y are tasks and $f_m(x)$ and $f_m(y)$ are the processor identifiers to which they are assigned. $F_v(x, y)$ is the amount of communication required between x and y, and $F_p(f_m(x), f_m(y))$ is the communication cost between the processors to which x and y are mapped. The product of these two quantities is calculated for each edge of the TIG.

Results were given for virtual 2D mesh applications on networks whose switches were interconnected in an arbitrary fashion. The assignments produced by two heuristics, $O(n \log n)$ switch-based mapping and $O(n^2 \log^2 n)$ binary mapping, were compared to those obtained by random mapping; up to 20% improvement over the random mapping was claimed for a 16×16 mesh.

In 2005, a nearly-identical objective function was proposed by IBM's Gyan Bhanot et al. [11]. This instance differs from the one above in machine topology and optimization method. Rather than analyzing irregular topologies, the IBM group chose to examine processors linked by a torus network. Like the instance above, however, communication costs vary significantly between different pairs of processors. For such networks, the difference in runtime between an efficient mapping and a poor one increases as the number of processors increases.

Computational load is assumed to be equal among all processors. In communication traffic matrix C, each entry $C(i, j)$ represents the amount of data communicated from domain i to domain j. In machine matrix H, each entry $H(i, j)$ represents the cost per unit of data to be sent between processor i and processor j, assumed to be the smallest number of hops on the BG/L torus between the two domains. Based on the ideas above, it was determined that the cost function for free energy, a term widely used in physics, should be

$$F = \sum_{i,j} C(i,j)H(i,j), \tag{2.4}$$

for all i, j such that domains i and j are mapped to processors i and j, respectively. The rationale is that minimizing such a function should minimize latency and link contention and thus communication time. A heuristic mapping of domains to processors was made, based on the H matrix, to provide an intelligent starting point. Simulated annealing was then applied to minimize the cost function.

On two applications on the Blue Gene/L, SAGE, an adaptive grid Eulerian hydrodynamics application, and UMT2000, an unstructured mesh photon transport problem, significant improvements were made in free energy over the default MPI rank order mapping. A shortcoming of this model is that achieving the lowest free energy does not necessarily correspond to achieving the minimum communication time in all cases, resulting from the relative simplicity of the cost function. Nonetheless, based on the ideas for mapping set forth by the model, IBM developed the Task Layout Optimizer for Blue Gene [46]. This software solution attempts to find the best mapping of MPI tasks to the Blue Gene/L processors, based solely on the communication matrix representing traffic between MPI tasks.

In 2006, Agarwal et al. [8] focused on decreasing link contention on torus and grid networks. They chose 'hop-bytes,' or the total size of inter-processor communication in bytes weighted by hop distance between the respective end-processors, as the metric for measuring the quality of a mapping. To minimize the objective function value, they employed a heuristic which, at each iteration, selects a task and a processor to which to map it based on a function that estimates for each pair of unallocated tasks and available processors the cost of placing the task on the processor in the next cycle. The task most critically in need of being placed in the next cycle is mapped to the processor for which the cost is lowest. For a 2D Jacobi-like benchmark, simulation results showed a reduction in hop-bytes and experimental results on Blue Gene/L showed a reduction in runtime. Simulation results were also provided for a molecular dynamics application.

The belief that TIG's are not descriptive enough for many problems led Dixit-Radiya and Panda to employ task precedence graphs (TPG's) [24]. They argued that TIG's capture volume of communication between tasks but ignore temporal information completely. They incorporated link contention and total traffic volume into their model for a small number of processors; the directed graph representation allows their model to identify communication steps that conflict temporally and spatially. The authors solved the problem with their min-max-contention heuristic, a strategy involving repeated pairwise exchanges of processors to minimize the maximum link contention. It is asserted that this heuristic may be applied to problems with both regular and irregular communication patterns and to any distributed-memory machine following shortest-path message routing. The authors suggested adapting their work for k-ary n-cubes, such as rings, hypercubes, and tori, with non-minimal adaptive routing.

Nicol's 1996 work with Mao [56] focused on problems exhibiting certain communication patterns. It is a more generalized version of earlier work which dealt with problems with a rectilinear topology; it looks into problems that can be represented with k-ary n-cube work and communication patterns and equal computation costs among all processors. The authors sought to find an assignment of tasks to processors yielding the smallest bottleneck for the specific graphs mentioned above. Bottleneck is measured as the maximal communication cost induced by the mapping among all processors. A processor's cost is generally measured as the sum of its computation and communication costs, but since computation costs are assumed to be equal for all processors in this model, the term 'bottleneck' here refers solely to communication costs.

2.1.2 Comprehensive models

Stone's classic 1977 work [63] is one of the earliest instances of task mapping literature. His original model is based on a two-processor heterogeneous system. Each task has a computation cost, and each message has a communication cost. The total cost of a mapping is the sum of all computation and communication costs. For $p = 2$, Stone showed that the problem could be reduced to a commodity flow problem and solved by the Ford-Fulkerson maximum flow algorithm in polynomial time. A graph was constructed with a source node corresponding to the first processor and a sink node corresponding to the second. Each task was represented by a node connected to both the source and sink; edge capacities represented the cost of executing the task on the first processor and second processor, respectively. The minimum cut corresponding to the maximum flow for this graph gives the optimal task mapping. Stone's model was extended to larger values of p, for which the problem was shown to be NP-complete.

The major shortcoming of Stone's model is that it is sequential. There is a linear ordering imposed on the tasks, so only one task may be executed at a time, on a single processor. This means that the sum of all computation and communication costs will be consistent with total time to solution, but the model will be useless for homogeneous collections of processors. In the case of a network of processors of identical computing power, it will always be optimal to assign all tasks to a single processor. Any other mapping would force unnecessary communication costs.

Bultan and Aykanat proposed a new mapping heuristic based on mean field annealing in their 1992 work [15]. They sought to minimize an objective function composed of two terms, the first representing communication and the second computation. This function is

$$H(s) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{K}\sum_{q\neq p} e_{ij}s_{ip}s_{jq}d_{pq} + \frac{r}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{K} s_{ij}s_{jp}w_iw_j, \tag{2.5}$$

where e_{ij} represents the amount of communication between tasks i and j, and s_{ab} represents the expected value of spin (a, b), quantifying the probability of mapping task a to processor b. The distance between processors p and q is represented by d_{pq}, w_a is the computational cost associated with task a, and r is a parameter introduced to balance the two objectives of the cost function.

It was shown that, on average, this heuristic slightly under-performs simulated annealing in communication cost, load imbalance, and overall solution quality, but drastically outperforms it in time to solution for the task mapping problem. No real applications were tested; rather, mean field annealing and simulated annealing were applied to various hypothetical cases using the objective function above. The problems consisted of TIG's with either 200 or 400 vertices of random weight and degree, and edges of random weight. The topologies chosen were 3-, 4-, and 5-dimensional hypercubes and 4×4 and 4×8 meshes.

Another comprehensive model was formulated in 1993 by Talbi and Muntean [66]. Both communication cost and load imbalance are taken into consideration; their relative weights can be adjusted through the parameter w in the cost function

$$F = C + wV. \tag{2.6}$$

C is identical to the full argument in (2.3) or (2.4) and is the communication cost; V is the variance of the loads of the different processors. Both of these components should be minimized.

Talbi and Muntean chose several benchmark problems and applied three main heuristics to find the best mappings: genetic algorithms, hill-climbing, and simulated annealing. They compared these heuristics, as well as a hybrid heuristic, on the basis of quality of solution and runtime. Each was found to have its advantages and disadvantages. Tabu search is suggested as another heuristic to adopt in future work.

In 2004, Salcedo-Sanz et al. proposed a comprehensive model for a heterogeneous system of processors of varying speeds [60]. They assumed knowledge of the maximum resources available for each processor to avoid overloading. They set forth the objective function

$$f(X) = \alpha_1 \sum_{j=1}^{M} t_j^e + \alpha_2 \sum_{i=1}^{N} \sum_{j=1}^{M} \sum_{p=1}^{N} \sum_{\substack{q=1 \\ q \neq j}}^{M} k_{ijpq} x_{ij} x_{pq} \tag{2.7}$$

subject to

$$\sum_{i=1}^{N} w_i x_{ij} \leq r_j, \qquad j = 1, 2, \ldots, M,$$

where α_1 and α_2 are adjustable parameters controlling the importance of each term in the cost function and $\alpha_1 + \alpha_2 = 1$. The variable M represents the number of processors, and N is the number of tasks. The total amount of time needed by processor j to finish its tasks is represented by

$$t_j^e = \sum_{i:x_{ij}=1} \frac{t_i}{v_j}, \tag{2.8}$$

where v_j is the speed of processor j relative to the speed of the slowest processor in the system. Each k_{ijpq} represents the communication cost between task i executing in processor j and task p executing in processor q. If task i has been assigned to processor j, $x_{ij} = 1$; otherwise, $x_{ij} = 0$. The goal is to minimize the objective function. The constraint ensures that the resource limit for each processor is not exceeded.

Salcedo-Sanz et al. tested different heuristics for arriving at an efficient solution; they include Hopfield neural networks (NN), genetic algorithms (GA), simulated annealing (SA), and hybrid NN/GA and NN/SA.

It may be observed that, in general, the objective functions of the comprehensive models are more detailed than those of the communication-only models. These more comprehensive models are consequently more difficult to manipulate and thus are often avoided. Whether this has a major impact on the effectiveness of the model depends on the problem and network topology at hand.

2.2 Solution Methods

After an objective function has been decided upon, a solution to the mapping problem for the given application and architecture is generally obtained by applying a heuristic. Exact algorithms are rarely employed due to their time complexity. Heuristics can often find near-optimal or even optimal mappings in only a fraction of the time.

2.2.1 Exact approaches

Exact approaches are not as commonly implemented, due to the long running time that they usually incur. Examples of algorithms which are used to obtain optimal solutions, rather than possibly suboptimal solutions as with the heuristic approaches, are branch and bound, dynamic programming, and weak homomorphism [65]. Since we want to extend our work to applications of a large size on thousands of processors, exact methods such as these are useless.

A possibility for making exact approaches feasible is to examine special instances of the mapping problem [57]. Adding assumptions of various types constrains the model, making it solvable in polynomial time.

2.2.2 Heuristic approaches

The heuristic approaches, many of which originate from the natural sciences, are all less time-consuming, but none guarantees an optimal solution.

Simulated annealing (SA), first proposed by Kirkpatrick et al. in 1983 [44], is a popular approach. Annealing is a process in which a substance is heated and then cooled slowly in order to lead it to reach its lowest energy state for the strongest structure [60]. SA adopts a similar strategy, but for numerical simulations. After an initial mapping is chosen, a neighboring mapping is selected before a cost comparison is made for determining whether the current mapping should be kept or replaced by the new one. If the new mapping translates into an improvement in cost, it is accepted. If it does not, it may still be accepted, but with a probability depending on the cost difference and the value of the 'temperature' variable, named after its counterpart in the true physical annealing process. Early in the algorithm the temperature is high, which allows relatively high probability of acceptance of the new mapping, even if it may appear farther away from the optimal state. As the simulation proceeds and equilibrium is reached at a given temperature, however, the temperature is reduced, making it less likely that a costlier new mapping will be accepted. The temperature changing scheme is called the 'cooling schedule.' Because it sometimes selects uphill, or more expensive, mappings, SA allows an efficient locating of a global extremum among suboptimal local extrema. It is widely used in applications such as the traveling salesman problem and the design of integrated circuits [58]. Disadvantages of SA are the sensitivity of its results to cooling schedule [54] and its relative slowness due to use of arbitrary pairwise exchanges [10]. However, it can easily be parallelized [11] to achieve speedup. An appropriate definition of neighborhood must also be determined.

The genetic algorithm (GA) is a stochastic search technique introduced by Holland in 1975 to imitate the biological evolution of a species [41]. As applied to task assignment, the individuals of a population represent possible mappings. Genetic operators are applied on the individuals. The steps involved in parallel GA are the following: generate a population of random individuals, assign a fitness value to each individual, and then repeat a cycle of select, reproduce, and replace a decided number of times. The term 'select' refers to making a list of pairs of individuals likely to mate, 'reproduce' refers to applying genetic operators (such as crossover and mutation) to the pairs, and 'replace' refers to swapping the worst individuals with better new ones, creating a new population [65]. Because the select phase is not purely deterministic, slight differences in fitness do not necessarily have a major impact on the survival of an individual. Mutation is a way to avoid stagnation. Just as natural evolution is inherently parallel, so is this approach [5].

Tabu search (TS) [38], developed in the 1970s, looks into successive neighborhoods to identify favorable moves, unlike simulated annealing which only looks one step ahead at a time. In addition, while SA only refers to temperature, TS uses multiple factors to determine whether or not to move to a new mapping. It records recent moves in order to keep track of areas of the solution space that have already been visited; it encourages new mappings to be explored by forbidding or heavily penalizing revisiting a mapping. This is where it obtains its name.

Neural networks (NN) are based on the interactions of neurons in the cerebral cortex [40]. They organize themselves such that stimuli from neighboring spots on the skin excite neighboring neurons, leading to short signal paths. As applied to task mapping, neurons become vertices and connections between neurons become edges. After each cycle of neuron updates, a check is done for convergence of the network. Convergence occurs when no neurons have changed their states during a cycle [60]. Kohonen networks are special types of neural networks often applied to task mapping [40].

Hybrid combinations of two or more of the heuristics described above, such as genetic simulated annealing (GA and SA) and mean field annealing (NN and SA), have also been developed [5, 15, 33, 37, 50, 60].

All of the heuristics above begin with an initial mapping in which all tasks have been mapped to processors and seek to improve upon it. Some other heuristics create a mapping one task at a time [10] so that only a partial mapping exists until the final iteration.

Chapter 3

Review of Supercomputers

The efficient assignment of tasks to processors depends greatly on the computing power of the processors as individuals and, more importantly, on the way in which these processors are linked together by their network. For this reason, we must fully understand the architecture, particularly the network, of any machine we plan to utilize.

3.1 Existing supercomputers

There are many different types of supercomputers, some of which we examine to allow sufficient understanding for experimentally testing our hypotheses. Our review highlights the aspects of the hardware relevant to building appropriate objective functions for our mapping of tasks. Processor clock speed, FPU (floating point unit) capabilities, size of memory, and processor-to-memory bandwidth affect single-processor computation time. The communication network topology and inter-processor latency and bandwidth affect communication time. All of these factors together, along with mapping, determine time to solution.

3.1.1 Beowulf systems: Galaxy

In 1994, the Beowulf Project was started at the Goddard Space Flight Center in Greenbelt, Maryland, under the sponsorship of NASA [53]. The goal of the venture was to design a commodity-based cluster system to serve as a cost-effective alternative to mainframes. The Beowulf clusters that exist today consist of any number of computers, possibly of several different types, connected through a switch network. Because of their ability to be built from commercial off-the-shelf components, Beowulf clusters provide low-cost solutions to parallel programming needs [53]. The downside to a Beowulf system is its relatively slow IPC (inter-processor communication) network with high latency, at 10 to 100 microseconds, and narrow bandwidth, at 1 to 10 Gbits/s.

Galaxy, housed in the Applied Mathematics Department at the State University of New York at Stony Brook, is the Beowulf system used for this work [1]. Galaxy utilizes a three-switch architecture and is composed of five types of nodes with different speeds. This heterogeneity is not common. All nodes of a given type reside on the same switch. Each node contains two processors; which processor gets assigned which piece of the problem or which processor communicates with which other processor is not inconsequential in terms of time to solution, and different mappings can produce significantly different runtimes due to the speed difference between intra-node and inter-node communication time.

There are two types of Galaxy nodes used for this work. Each node of the first type (a1533) has two AMD Athlon XP 1800+ processors, running at 1.5 GHz, with a 128 KB L1 cache and a 256 KB L2 cache; additionally, each node has 1 GB of RAM. Each node of the second type (p3200) holds two Intel Xeon processors, running at 3.2 GHz. ADD MEMORY INFO HERE TO BE COMPLETE

3.1.2 Distributed SMP systems: IBM pSeries 655

Distributed SMP (symmetric multiprocessing) systems are designed to house 2 or more (for example, 512) processors on a single node. Several levels of cache are utilized; some memory is distributed and some is shared among various of processors, resulting in a very well-defined, non-uniform memory access platform.

The distributed SMP system utilized for this thesis work is an IBM pSeries 655 supercomputer awarded to Stony Brook University in 2005 through IBM's SUR (Shared University Research) Program. The basic computing component of the pSeries system is the Power4 chip. A single chip contains two independent processors, each having its own 32 KB L1 data cache. A 1.5 MB L2 cache is shared by the two processors. Up to four Power4 chips can be joined to form a multi-chip module (MCM) of as many as eight processors. Four such eight-processor modules can then be linked to form a 32-way system. Stony Brook's pSeries machine, housed at BNL, consists of four modules, each containing eight processors running at 1.66 GHz, making a speed of up to 6.64 Gflops per processor possible. Each node of eight processors shares a

128 MB L3 cache, in addition to 8 GB of shared local RAM. The connection method between processors within a node is bus-based; the connection method among nodes is switch-based [14]. The communication distance between any pair of processors on the same node is equal, with low-latency, high-bandwidth connections. Higher latency is found when processors on different nodes communicate, because of the switch involved.

3.1.3 Cellular architectures: Blue Gene/L and QCDOC

The cellular architecture style of design resulted from the need for faster access to memory by individual processors and scalability after massive system expansion [48]. Consequently, all cells can search their own memories for data, rather than having a central controller deciding which ones should do it. The focus is on quickly accessing, searching, and moving data. In order to reap the full benefits of cellular architectures, the systems should consist of tens of thousands of processors. Each cell is either a single chip or several cores, and communication cost is reduced by linking a cell to only its nearest neighbors, or those cells physically closest to it. Latency is commonly as low as .1 microseconds, and bandwidth as high as 200 Gbits/s. In addition, if a cell fails, data can be rerouted to avoid the dead-link problem [48]. Processors in these systems are often described as "slow but cool." The goal is to achieve decent processor speed with low power consumption. Blue Gene/L (BG/L) and QCDOC are the cellular architectures utilized for this work.

The main communication network of BG/L is a three-dimensional (3D) nearest-neighbor torus network. Employing IBM's system-on-a-chip technology, each BG/L node consists of a single ASIC containing two embedded 700 MHz IBM PowerPC 440 processors (each with an attached 128-bit FPU operating at 2.8 Gflops), 4 MB of shared EDRAM, a controller for an external 512 MB shared DDR SDRAM, and a controller for each of the three communications networks. Each processor has a 32 KB instruction cache and a 32 KB data cache. In summary, BG/L is a power-efficient solution designed for simulation of physical phenomena, real-time data processing, and offline data analysis [29].

Compute nodes on BG/L are logically arranged into a 3D lattice, and the torus network connects nearest neighbors in the lattice. Because of the wraparound nature of the torus, this means that every node is linked directly to six neighboring nodes [29]. BG/L allows point-to-point routing [52]. Each node supports six two-way communication links, one for each of the two directions in all three dimensions. Total bandwidth per node is 2.1 GB/s. Latency is approximately 100 ns [29].

BG/L has an additional network for global communication, a global tree. The torus is the primary communications network, handling both point-to-point messaging and many global or collective operations. The tree is used solely for those collective communications for which the torus network is inefficient. BG/L also has a double floating point unit, enabling BG/L to perform two fused multiply-adds per cycle. In terms of memory, BG/L's bandwidth from EDRAM to CPU is 22 GB/s [52].

BG/L nodes may operate in one of two modes: communication coprocessor mode or virtual node mode [29]. Communication coprocessor node is the default; one processor runs the main thread of the compute process while the other handles primarily communication. In virtual node mode, node resources are split evenly between the two processors, and each acts as a separate entity running its own process.

We gratefully acknowledge use of a 1024-node BG/L system operated by the Argonne Leadership Computing Facility at Argonne National Laboratory (ANL) in Argonne, IL. Access to 32- ($4 \times 4 \times 2$) and 64-node ($8 \times 4 \times 2$) partitions can be obtained without assistance; access to 128- ($8 \times 4 \times 4$), 256- ($8 \times 4 \times 8$), 512- ($8 \times 8 \times 8$), and 1024-node ($8 \times 8 \times 16$) configurations can be obtained by request. Each node consists of two PowerPC 440 processors running at 700MHz. Each has links to its six nearest neighbors; the bandwidth of each link is approximately 150 MB/s. Currently, up to 64 nodes have been used. Only communication coprocessor mode has been used thus far. This was the original choice of mode because it enabled the notion of hop count to be used for all processor pairs.

As their name suggests, QCDOC (quantum chromodynamics on a chip) supercomputers were designed to model quantum chromodynamics problems. These problems deal with the theory of the strong nuclear force and are usually modeled as 4D or possibly 5D lattices [32]. QCDOC is a multiple-instruction, multiple-data (MIMD) machine with distributed memory. Work began on designing the QCDOC ASIC (application specific integrated circuit) in 1999. The ASIC contains an embedded 500 MHz IBM PowerPC 440 processor with an attached 64-bit FPU operating at 1 Gflops, 4 MB of embedded dynamic random access memory (EDRAM), a controller for external double-data-rate synchronous DRAM (128 MB DDR SDRAM), and a serial communications unit (SCU) controlling the transmission of data to and from each of a node's 12 neighbors. The EDRAM bandwidth is approximately three times as large as that of the DDR SDRAM (8 GB/s versus 2.6 GB/s) [30, 31]. Each processor has a 32 KB instruction cache and a 32 KB data cache. QCDOC incorporates all of the features of its predecessor, QCDSP, and more, on a single chip, using IBM's system-on-a-chip (SoC) technology. QCDOC boasts low latency and low power consumption [19, 30, 32, 43].

The main QCDOC communication network, and the one relevant to this work, is the truncated 6-dimensional (6D) nearest-neighbor torus network. Three of the dimensions of the 6D torus are closed on a motherboard and therefore are of maximum size two, and the others are open to off-board communication and can be of any size [23]. In addition, the machine can be remapped to form partitions of a lower dimension. Machines which support MPI allow a similar process; however, with MPI, the remapping occurs at runtime [32]. The QCDOC nearest-neighbor serial communications unit has an approximate latency of 600 ns for a 64-bit send/receive. Each node's SCU controls 24 off-node simultaneous bit-serial communication links (one for send and one for receive in each of the directions in each of the six dimensions). Total nearest-neighbor bandwidth is 1.3 GB/s at 500 MHz, utilizing simultaneous sends and receives. For normal data transfers, the "three in the air" protocol applies [31]. This refers to three 64-bit data words being sent before an acknowledgement is given.

While the other architectures examined utilize MPI, QCDOC utilizes

QMP for message passing between processors. QMP is a message passing library developed by the LQCD (lattice quantum chromodynamics) research community. It was designed with LQCD applications in mind, so its strong point is fast nearest-neighbor communication [3]. Global operations are implemented using the store-and-forward method with a predetermined path, unlike the adaptive cut-through routing of Blue Gene/L. The implementation of QMP on QCDOC does not contain non-nearest neighbor communications. In addition, the upper limit on message size is 1.5 MB.

There are two QCDOC machines available for use for this project, one at Brookhaven National Laboratory (BNL) in Upton, NY, and one at Columbia University in New York, NY. Access to 1-, 2-, 4-, 8-, and 16-node partitions at Columbia and access to a 64-node ($2 \times 2 \times 2 \times 2 \times 2 \times 2$) network at BNL can be obtained without assistance; access to larger configurations of the form $2 \times 2 \times 2 \times K \times M \times N$ can be obtained by request. Currently, up to 64 processors have been used. Potentially, up to 8,192 can be used.

As seen above, there are both similarities and differences between the Blue Gene/L (BG/L) and QCDOC supercomputers. Both machines are massively parallel cellular architectures, scaling up to tens of thousands of nodes. Their main communication networks are nearest-neighbor torus networks employing IBM's system-on-a-chip technology. However, these tori differ in dimensionality. While QCDOC supports only nearest-neighbor communication, BG/L allows point-to-point routing [52]. BG/L has an additional network for global communication, a global tree. It also has a double floating point unit, as opposed to QCDOC's single floating point unit. In terms of memory, BG/L's bandwidth from EDRAM to CPU is almost three times as large as QCDOC's [52]. For these reasons, it is interesting to examine both of these types of cellular architectures as individual machines.

3.2 Possibilities for the future

In addition to examining existing supercomputer architectures, a goal for the future is to consider new network topologies that have not been implemented yet, ones much more complex and difficult to visualize than the QCDOC or BG/L tori. For example, we can imagine an architecture whose base level is a $4 \times 4 \times 4$ structure, where each of the 64 vertices is actually another $4 \times 4 \times 4$ structure, each of whose vertices is another $4 \times 4 \times 4$ structure. Ultimately, in this case, we have $64 \times 64 \times 64$ nodes. The intuitive notion is that of a tree or some other hierarchical structure. Such architecture may be proven to be more efficient than tori in the future, for a class of applications in services necessitating complicated communication patterns rather than scientific applications with more regular patterns. It may be found that relationships between the two different structures exist, however, and techniques that work on one may be applicable in some way to the other, with modifications. This is a type of isomorphism. Proper mathematical representation and analysis of the properties of such architecture is new and challenging.

Work by Suri and Mendelson [64] has been done to propose future architectures based on communication patterns of applications.

When constructing a new supercomputer, there are many balanced architectural aspects to consider. One is how to best connect the nodes topologically at high bandwidth and low latency. Conclusions about task assignment drawn from this thesis will aid in the efficient design of such machines.

Chapter 4

Case Studies

The task mapping scheme depends highly on the architectures or applications. In a star network consisting of a homogeneous collection of processors with equal network distance, all load-balanced mappings are equally efficient. The same holds true for simple problems in which each task must communicate in an identical way with every other task.

Such simple instances described above are rare in actual applications. Even in a case where only all-gather communication is involved, all load-balanced mappings are not equally efficient. Although all processors need to receive information from all other processors, all processors' communication demands cannot be viewed as isomorphic because the communication occurs in steps, and some tasks never communicate directly with some other tasks. Figure 4.1 provides an illustration of this idea. In almost all cases, we can find significant improvement in runtime between implementations based on smart mappings and those not.

In order to demonstrate the significance of task mapping, we compare

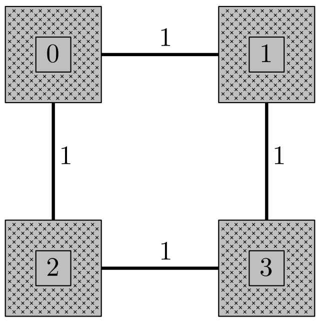

Figure 4.1: Processor arrangement and network distance for all-gather example

timing results achieved with implementations of various mappings. Among the possibilities for comparison to the automatic mapping determined by a given model are the following:

ROMap (the rank order mapping) is obtained by mapping task i to the processor of rank i. This type of mapping is used often in practice because it is the default that occurs when no effort is made to assign tasks to processors efficiently. For cases in which MPI is used for message passing, the `MPI_Comm_rank` routine assigns a rank, or ID, to each processor. Similarly, for cases in which QMP is used, `QMP_get_node_number` makes the assignment.

MMap (a manual mapping) is a human's best attempt at a good mapping using pencil and paper. Manual mapping is also common.

RMap (a random mapping) is obtained by randomly assigning tasks to processors. When we present results, times for RMap represent the average of 100 random maps. We would ideally like to give the true expected value of a random map.

IPMap (an intentionally poor mapping) is a human's best attempt at a poor mapping using pencil and paper. It is always possible to make an extremely inefficient mapping by creating great load imbalance in using the processors and network resources. For applications for which load balance is easy to achieve, however, IPMap has been chosen to be a load-balanced mapping with high inter-processor communication costs.

Of the assignments above, the automatic mapping determined by the given model should correspond to the shortest runtime. If this is not the case, we can conclude that the model at hand is flawed. The runtimes for the implementations based on MMap and ROMap should be the next best, but should still be significantly longer than those for the mapping determined by the given model. In the average case, RMap should correspond to the second-longest runtime. Finally, IPMap should have the highest runtime. These last two are not normally used in practice; they are used in our work as worst-case mappings to compare to our best-case mapping determined by the given model.

We examine the consistency of the best mapping determined by our model with the experimentally best mapping. All mappings are implemented manually. The timing results obtained can be compared to results produced by alternative mappings.

4.1 Our models

Following the work of [8, 10, 11, 24, 47, 55, 62], we study applications for which load balance is naturally achieved on a homogeneous system. Many decomposition methods for many applications may lend themselves naturally to perfect load balance. With this balanced computing load, our mapping model only needs to take into account the communication portion of the cost without having to include computation. Additionally, our primary goal is ordering the mappings in terms of runtimes, so we can avoid the additional cost required in incorporating computation into the model.

We assume that the number of tasks is equal to, or an integer multiple of, the number of processors [11, 24]. The first model we test is a communication-only model similar to those in [8, 11, 55]. The machine is represented by its supply matrix, S. The matrix multiplication application is represented by its demand matrix, D. We let P be the set of processors and T be the set of tasks. The mapping is a function $f : T \to P$. The objective function adopts a QAP form. The goal is to minimize the cost of the mapping

$$C = \sum_{t_i, t_j \in T} S(f(t_i), f(t_j)) D(t_i, t_j). \tag{4.1}$$

In terms of the communication, we prove in 4.2.5 that it is possible to make a mapping for which we can guarantee that collision is avoided for matrix multiplication on QCDOC; we have not found such a mapping for all applications on all four machines. Therefore, failing to incorporate link contention into the model may decrease its value in some cases. Because it greatly complicates the model, and consequently lengthens the solution process, however, we choose to ignore it in this initial case. In addition, past research which has taken link contention into consideration has usually incorporated it through the steps of a heuristic algorithm to optimize the mapping, rather than through additional terms in the objective function itself [24, 67]. We should also be aware that link contention does not necessarily translate into an increase in runtime [24]. Messages attempting to use the same link are not always the source of a bottleneck.

We look at a second model similar to the first except for a slight modification in objective function. Rather than the network distance between processors, we use the square of the distance. When choosing a metric like hop distance to quantify the supply matrix, we assume that communication time varies directly as the number of hops between processors. This may lead to inaccuracy because it ignores the possible additional time for traversing intermediate nodes and does not consider the fact that increased distance between processors may make collision more likely. This rationalizes the relevance of examining this modification.

It may not be intuitively obvious that using such a metric rather than the distance can change the ordering of the mappings. In Figure 4.2 we use a simple example of 4×4 supply and demand matrices to illustrate the rationale. If we map tasks to processors using ROMap, or tasks (0, 1, 2, 3) to processors (0, 1, 2, 3), respectively, the objective function value using distance is $1(2) + 1(2) + 1(2) + 1(2)$, or 8. If we map tasks (0, 1, 2, 3) to processors (1, 0, 3, 2), respectively, the objective function value is $1(1) + 1(4) + 1(1) + 1(1)$, or 7. The

$$\begin{array}{c} \text{Demand} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c} \text{Supply} \\ \begin{bmatrix} 0 & 2 & 2 & 1 \\ 1 & 0 & 2 & 4 \\ 1 & 1 & 0 & 2 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Figure 4.2: Demand and supply matrices for comparing objective functions

second map produces a better objective function value. If we look at distance squared, however, ROMap gives a value of $1(4) + 1(4) + 1(4) + 1(4)$, or 16, and the second map gives a value of $1(1) + 1(16) + 1(1) + 1(1)$, or 19. ROMap produces a better objective function value in the distance squared case. It is interesting to note that neither of these mappings is optimal with respect to objective function value for either objective function, however. The optimal mapping is the same for both objective functions: assigning tasks (0, 1, 2, 3) to processors (3, 2, 1, 0), repsectively, gives the optimal value, 4 in both cases.

Although we examine this as a change in objective function, we note that it is equivalent to squaring all entries in the supply matrix and using the objective function from the first model. In our analysis, we choose to square the entries in the supply matrix rather than square the term in the objective function for the added value of reducing the computational time required to optimize the mapping.

Other related modifications are obvious at this point. The term representing volume of communication in the objective function may be squared, and the distance term left as is. This is equivalent to squaring all entries in the demand matrix. This may be appropriate if the number of sends is not an accurate representation of the volume of communication between tasks. Along the same lines, the entire objective function argument from our first model may be squared, representing a squaring of all entries in both the supply and demand matrices. It is worth noting that multiplying the objective function by a constant value cannot change the ordering of mappings, so we do not study this modification.

After selecting an objective function, the next step is to obtain a solution in the form of an efficient mapping. Because there are $n!$ possible mappings, where n is the number of tasks, we should not examine all possibilities to guarantee an optimal solution, even for a relatively small number of tasks. Instead, we consider heuristics to apply to minimize the cost function. Following closely related work [11], we have chosen simulated annealing. Although this heuristic is slower than most, it can easily be parallelized to reduce runtime. In addition, we assume that most applications will run repeatedly after an efficient mapping is discovered, so the expected gain in solution quality is worth the expected increase in heuristic runtime. This completes the theoretical argument. Then the mapping is implemented manually, and timing results are obtained and compared to results produced by alternate mappings. A conflict in the theoretical and experimental data suggests flaws in the model. The most likely cause of discrepancy between the theoretical and the experimental would be an objective function which did not adequately measure time to solution, either because it was too simple or did not correctly merge the factors it utilized. We have applied this process to Galaxy, the IBM pSeries 655, and BG/L. The process differs slightly for QCDOC due to its lack of non-nearest neighbor communication.

We then go through a similar process for subsequent applications. We first complete the MMMI phase and then incorporate models into the project to move on to the AMMI phase. With each completed application, we have more insight into task mapping as a whole. We can then move away from the specific and make statements which can be more widely applied about the validity of the model and even the entire approach.

4.2 Matrix multiplication

Matrix multiplication is a common problem in many scientific and engineering applications, especially linear algebra applications such as inverting matrices, solving systems of linear equations, and finding determinants [59]. Serial matrix multiplication is simple, as it requires only a few lines of code. Implementing it on a parallel machine, however, brings up many debatable points. Matrix multiplication is computation-intensive [36], so we cannot expect to find a drastic difference in overall runtime between a good mapping and a bad one for a given parallel algorithm. However, since there is still communication involved, improvement is significant when a mapping determined by an appropriate model is applied. We address applications which benefit immensely from intelligent task mapping later in this chapter.

4.2.1 Implementation history

Standard sequential multiplication of two $n \times n$ matrices requires $O(n^3)$ multiplications. In 1969, Strassen produced an algorithm requiring only $O(n^{\log_2 7})$, or approximately $O(n^{2.807})$, multiplications. Coppersmith and Winograd reduced the exponent further to approximately 2.376 in 1987. Popular belief is that the theoretically best algorithm can be completed in essentially $O(n^2)$, the lower bound [59]. Cohn, Kleinberg, Szegedy, and Umans take a group theoretic approach in their 2005 work [21] and set forth two conjectures; if either is shown to be correct, the exponent will be proven to be 2. Theoretical advances do not necessarily translate into improvements in runtime, however. These discoveries are often impractical to implement because they may apply only to specific cases or their methods may be extremely intricate. In addition, the constant terms in front may be so large that these methods only become faster for unrealistically large matrix sizes.

In an effort to decrease the actual runtime of serial matrix multiplication, as well as other sequential routines from the BLAS (basic linear algebra subprograms) library, an optimization package called ATLAS (automatically tuned linear algebra software) was created [13, 69, 70]. Among other things, it improves upon the BLAS routine GEMM (general matrix multiply).

Our focus, however, is not on determining the best serial matrix multiplication method or even the best parallel method. Instead, we wish to find an efficient mapping of tasks to processors for a given parallel method. We adopt the standard sequential multiplication method for the serial portion of our code. A better serial method will lead to a decrease in runtime for all mappings, but the relative efficiency of the various mappings will not change.

When we discuss parallel matrix multiplication, we must mention data decomposition as well as the algorithms themselves. In block distribution, data elements are divided into blocks of consecutive elements which are then allocated, in order, to consecutive processors. For convenience, the size of the blocks is chosen so that the elements do not wrap around the processor array. In cyclic distribution, the first element is allocated to the first processor, the second to the second processor, etc. If there are more elements than processors, the distribution wraps around the processor array until all elements have been allocated. Cyclic distribution is often useful for spreading the computational load evenly over processors. In block cyclic distribution, blocks of consecutive elements are allocated cyclically to processors.

In many programs employing one-dimensional block decomposition, the A matrix is blocked by row and the B matrix by column. A block of A, consisting of $\frac{n}{p}$ rows of length n, and a block of B, consisting of $\frac{n}{p}$ columns of length n, are distributed to the same processor. Throughout the course of the algorithm, each processor retains its block of B. At each iteration, all processors simultaneously perform serial matrix multiplication; then each processor passes its block of A to the processor above it. There are many implementations, and the one we have chosen is known to be inferior to Fox's method, which is based on a two-dimensional decomposition. We will compare the experimental differences in runtime between the two methods.

Two classic parallel algorithms, both based on two-dimensional block decomposition of the A and B matrices, are Cannon's 1969 systolic algorithm [16] and Fox's 1987 broadcast-multiply-roll algorithm [34]. Both require a square number of processors, $p = q^2$, and involve decomposition of the matrices into $q \times q$ grids. The A and B matrices are blocked by both row and column.

Each processor begins with an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of A and an $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ block of B.

Cannon's algorithm is systolic because of the regular pattern of sending data between directly-connected processors only. Each processor, at each step, performs serial matrix multiplication and then passes its block of A to its neighbor to the left in the row and its block of B to its neighbor above in the column. This is repeated until each processor has seen all blocks of A and B necessary to obtain a block of the result matrix C.

In Fox's broadcast-multiply-roll algorithm, the broadcast step pertains to the A matrix. One block in each row is broadcast to the processors to which the other blocks in the row have been mapped. In the multiply step, each processor performs serial matrix multiplication: the block of A just received is multiplied by the currently-stored block of B. The roll step pertains to the B matrix. Each processor sends its block of B to the processor above it. This is the main implementation we have chosen. It was chosen over Cannon's algorithm largely because the six-dimensional network nature of the QCDOC supercomputer allows for a fast broadcast.

In recent years, researchers have attempted to address the shortcomings of these algorithms. Both PUMMA [18] and BiMMeR [42] were based on variants of Fox's broadcast-multiply-roll method. The key contribution of these packages was their universality; they allowed matrix multiplication to be performed for general 2D processor grids, rather than for square grids alone. The published results for both packages were obtained from runs on an Intel Touchstone Delta [18, 42].

The main difference between PUMMA and BiMMeR was their choice of data distribution. PUMMA (parallel universal matrix multiplication algorithm) was implemented by Choi, Dongarra, and Walker in 1994, using a 2D block cyclic data distribution. BiMMeR was implemented by Huss-Lederman, Jacobson, Tsao, and Zhang in the same year, using a virtual 2D torus wrap data layout. These differences in data distribution led to algorithmic differences.

In their 1994 paper, Agarwal et al. set forth the new broadcast-broadcast approach [7]. A similar approach was attempted independently by van de Geijn and Watts, leading to the implementation of SUMMA (scalable universal matrix multiplication algorithm) [68]. The contributions of these two groups were simpler algorithms yielding better performance and requiring less space, using improved overlapping of computation and communication. The published results were obtained from runs on the Intel Paragon mesh system. Grayson et al. used SUMMA on an Intel Paragon system in 1996 to obtain a high performance parallel implementation of Strassen's algorithm [35].

Agarwal et al. presented an approach using a 3D data distribution that required less communication than its 2D counterpart in their 1995 paper [6]. They addressed the previously unstudied case of minimizing communication for matrices of arbitrary shape, and they implemented their code on IBM POWERparallel SP2 multistage network systems.

In 1998, Choi built on SUMMA and created DIMMA (distribution-independent matrix multiplication algorithm) for block cyclic data distribution [17]. His two contributions were adding a modified pipelined communication scheme to overlap computation and communication more efficiently and finding a way to maintain the maximum performance of the sequential BLAS matrix multiplication routine, DGEMM (double-precision general matrix multiply), even for extremely small or large block sizes. Implementation was on the Intel Paragon.

This work is relevant because it provides information about fast implementations of matrix multiplication. However, the focus of the literature discussed above is not on task mapping; rather, it is on effective task decomposition and optimizing code from a computer hardware structure approach. Smart memory access and efficient use of cache are looked to as the means by which performance can be improved. When task assignment is mentioned in these works, there is usually a natural mapping of sub-matrices to processors, so models and comparisons of the solutions they give are not discussed.

Krishnan and Nieplocha's 2004 work on matrix multiplication on clusters and shared memory systems [45] puts more emphasis on task assignment. The authors' approach on these types of machines is to avoid message passing; instead, they make explicit use of shared memory and remote memory access (RMA) communication. They tested their approach, SRUMMA (shared and remote-memory based universal matrix multiplication algorithm), on IBM SP and Linux-Myrinet clusters, as well as SGI Altix and Cray XI shared memory systems. The growing trend to improve cost-effectiveness by using symmetric multiprocessing (SMP) nodes as building blocks of parallel systems is cited as the main reason for this new direction of research. Because of their use of different strategies for communication depending on whether data is in the shared memory domain of two processors or in an unshared memory domain, task mapping comes into play.

We have implemented variations of matrix multiplication programs utilizing both one-dimensional and two-dimensional block decompositions on Galaxy, SUR, BG/L, and QCDOC, using the two algorithms described above. The programming language used is C++. QMP has been employed for message passing on QCDOC, while MPI has been used on the other three systems.

We describe the row headings of the tables in the following matrix multiplication subsections:

- Objective Function Value: This is the model's objective function value for a given mapping.

- Normalized Obj. Fc'n Val. The normalized objective function value is the given objective function value divided by the best objective function value. Using this measure, we can view relative improvement more easily.

- Experimental Comm. Time: The experimental communication time is the time the actual application code spends in communication.

- Normalized ECT: The normalized experimental communication time is given with respect to the experimental communication time for the best mapping.

- Experimental Total Time: This is the total runtime for the application code.

- Normalized ETT: The normalized experimental total time is given with respect to the experimental total time for the best mapping.

We describe the column headings of the tables in the following matrix multiplication subsections:

- Best: This is the best mapping, as determined by simulated annealing on the model's objective function.
- ROMap: This is the rank-order mapping, which is the default mapping in most cases.
- Alternate: This is a mapping chosen by hand to be inefficient.

4.2.2 Matrix multiplication on Galaxy

The theoretical supply matrix for Galaxy consists of ones for pairs of processors on the same node and some constant $\beta > 1$ for pairs of processors on different nodes. $\beta > 1$ is the same for all inter-node pairs because of the switch nature of the network.

In the experimental supply matrix for Galaxy, the entries are

$$c_{i,j} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are on the same node;} \\ 2.76, & \text{otherwise (different nodes).} \end{cases}$$

The 2.76 represents $\frac{1.26}{.456}$; the conversion is made to give intra-node communication a unit cost.

For both the 1D and 2D decompositions, our simulated annealing heuristic finds the optimal mappings.

1D block decomposition: For matrix multiplication using the 1D block decomposition, we order our tasks from 0 to n from top block to bottom block in the matrix multiplication matrix. The optimal mapping is to assign the first two tasks to two processors on the same node, the next two tasks to two processors on another node, etc. This is clearly optimal because all communications are of equal magnitude and this mapping achieves the maximum possible number of intra-node communications. For our chosen task numbering, this is also the default, or rank-order, mapping. The alternate (intentionally poor) mapping chosen is to avoid assigning consecutive tasks, i.e., those requiring more pairwise communications than others, to processors on the same node. In this case, there is no use of intra-node communication; only inter-node communication occurs.

Mapping	**Objective Function Value**
Optimal	$\frac{p}{2}\left(p-1\right)\left(2.76+1\right)$
Alternate	$p\left(p-1\right)\left(2.76\right)$

Table 4.1: Objective function values for 1D block decomposition on Galaxy

We give the general form of the objective function values for all even $p \geq 4$ for the optimal and alternate mappings in Table 4.1. For the optimal mapping, half of the processors send to another processor on the same node and half send to another processor on a different node at each iteration. There are $p-1$ iterations, so we have

$$\left(\frac{p}{2}\left(1\right)+\frac{p}{2}\left(2.76\right)\right)\left(p-1\right)$$

This simplifies to the expression in the table. The amount of data sent in each communication is the same for all messages.

For the alternate mapping, each of the p processors sends at each of the $p-1$ iterations to another processor on a different node; this leads directly to the expression in the table.

	Best	**ROMap**	**Alternate**
Objective Function Value	451.2	451.2	662.4
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	0.0913	0.0913	0.163
Normalized ECT	1.00	1.00	1.79
Experimental Total Time	0.887	0.887	0.934
Normalized ETT	1.00	1.00	1.05

Table 4.2: Galaxy, N=1024, $P = 16$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	1864.96	1864.96	2737.92
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	0.233	0.233	0.347
Normalized ECT	1.00	1.00	1.49
Experimental Total Time	0.649	0.649	0.712
Normalized ETT	1.00	1.00	1.10

Table 4.3: Galaxy, N=1024, $P = 32$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	451.2	451.2	662.4
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	0.388	0.388	0.616
Normalized ECT	1.00	1.00	1.59
Experimental Total Time	5.99	5.99	6.13
Normalized ETT	1.00	1.00	1.02

Table 4.4: Galaxy, N=2048, $P = 16$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	1864.96	1864.96	2737.92
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	0.549	0.549	0.794
Normalized ECT	1.00	1.00	1.45
Experimental Total Time	3.36	3.36	3.59
Normalized ETT	1.00	1.00	1.07

Table 4.5: Galaxy, N=2048, $P = 32$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	451.2	451.2	662.4
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	0.834	0.834	1.40
Normalized ECT	1.00	1.00	1.68
Experimental Total Time	19.83	19.83	20.32
Normalized ETT	1.00	1.00	1.02

Table 4.6: Galaxy, N=3072, $P = 16$, 1D block decomposition

Objective function value does not depend on problem size, and normalized objective function value does not depend on problem size or number of processors. If we make this choice, we should verify that experimental results follow the same trend.

For a given problem size, we see a decrease between the two mappings in all cases with respect to relative difference in experimental communication time. However, we see an increase in all cases with respect to relative difference in experimental total time. Since we want to minimize time to solution, we are more concerned with experimental total time, so this discovery does not present a problem.

2D block decomposition: For matrix multiplication using the 2D block decomposition, we order our tasks from 0 to n as we traverse the matrix

	Best	**ROMap**	**Alternate**
Objective Function Value	1864.96	1864.96	2737.92
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	1.01	1.01	1.60
Normalized ECT	1.00	1.00	1.58
Experimental Total Time	10.42	10.42	10.94
Normalized ETT	1.00	1.00	1.05

Table 4.7: Galaxy, N=3072, $P = 32$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	451.2	451.2	662.4
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	1.44	1.44	2.48
Normalized ECT	1.00	1.00	1.72
Experimental Total Time	46.66	46.66	47.63
Normalized ETT	1.00	1.00	1.02

Table 4.8: Galaxy, N=4096, $P = 16$, 1D block decomposition

multiplication matrix from top left block to bottom right block, traversing each row from left to right. The optimal mapping is to assign pairs of vertically adjacent tasks (vertically adjacent sub-matrices) to pairs of processors on the same node. The alternate (worst-case) mapping chosen is to assign tasks to processors in a way such that no adjacent sub-matrices are mapped to processors on the same node. In this case, there is no intra-node communication; only inter-node communication occurs. The default mapping falls between the optimal and worst-case mappings.

We give the general form of the objective function values for the default, optimal, and alternate mappings for all square $p \geq 4$ in Table 4.10. For the default mapping, each processor broadcasts to one processor on the same node and $\sqrt{p} - 2$ processors on other nodes. At each iteration, each processor rolls

	Best	**ROMap**	**Alternate**
Objective Function Value	1864.96	1864.96	2737.92
Normalized Obj. Fc'n Val.	1.00	1.00	1.47
Experimental Comm. Time	1.58	1.58	2.66
Normalized ECT	1.00	1.00	1.68
Experimental Total Time	24.11	24.11	25.16
Normalized ETT	1.00	1.00	1.04

Table 4.9: Galaxy, N=4096, $P = 32$, 1D block decomposition

Mapping	**Objective Function Value**
Default	$p\left(1 \times 1 + \left(\sqrt{p}-2\right)2.76 + \left(\sqrt{p}-1\right)2.76\right)$
Optimal ($p \neq 4$)	$p\left(\sqrt{p}-1\right)2.76 + \frac{p}{2}\left(\sqrt{p}-1\right)2.76 + \frac{p}{2}\left(\sqrt{p}-1\right)1$
Optimal ($p = 4$)	$p\left(\sqrt{p}-1\right)2.76 + p\left(\sqrt{p}-1\right)1$
Alternate	$2p\left(\sqrt{p}-1\right)2.76$

Table 4.10: Objective function values for 2D block decomposition on Galaxy

its block up to a processor on a different node. There are $\sqrt{p}-1$ iterations, so we have

$$p\left(1\left(1\right) + 2.76\left(\sqrt{p}-2\right) + 2.76\left(\sqrt{p}-1\right)\right)\left(\sqrt{p}-1\right).$$

This simplifies to the expression in the table.

For the optimal mapping, each processor broadcasts to $\sqrt{p}-1$ processors on other nodes. At each iteration, half of the processors roll their blocks up to processors on different nodes but half roll them up to a processor on the same node. For $p \neq 4$ this gives us

$$p\left(\sqrt{p}-1\right)(2.76) + \tfrac{p}{2}\left(\sqrt{p}-1\right)(2.76) + \tfrac{p}{2}\left(\sqrt{p}-1\right)(1).$$

For the alternate mapping, each processor sends only to processors on different nodes. For each processor, there are $\sqrt{p}-1$ such sends for the iteration in which the processor broadcasts. Each processor also rolls its block up in each of the $\sqrt{p}-1$ iterations, so we have

$$p\left(\left(\sqrt{p}-1\right)2.76+\left(\sqrt{p}-1\right)2.76\right).$$

This simplifies to the expression in table 4.10.

	Best	**ROMap**	**Alternate**
Objective Function Value	222.72	236.8	264.96
Normalized Obj. Fc'n Val.	1.00	1.06	1.19
Experimental Comm. Time	0.0552	0.0736	0.0785
Normalized ECT	1.00	1.33	1.42

Table 4.11: Galaxy, N=1024, $P = 16$, 2D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	222.72	236.8	264.96
Normalized Obj. Fc'n Val.	1.00	1.06	1.19
Experimental Comm. Time	0.197	0.254	0.282
Normalized ECT	1.00	1.29	1.43

Table 4.12: Galaxy, N=2048, $P = 16$, 2D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	222.72	236.8	264.96
Normalized Obj. Fc'n Val.	1.00	1.06	1.19
Experimental Comm. Time	0.436	0.547	0.642
Normalized ECT	1.00	1.25	1.47

Table 4.13: Galaxy, N=3072, $P = 16$, 2D block decomposition

Although the decreases in overall runtime may not appear high, we must keep in mind that matrix multiplication is a computation-intensive application.

	Best	**ROMap**	**Alternate**
Objective Function Value	222.72	236.8	264.96
Normalized Obj. Fc'n Val.	1.00	1.06	1.19
Experimental Comm. Time	0.766	0.991	1.14
Normalized ECT	1.00	1.29	1.49

Table 4.14: Galaxy, N=4096, $P = 16$, 2D block decomposition

Changing the mapping affects only communication time here because we have already achieved perfect load balance. When looking at communication time only, we consistently see significant improvement.

Even when using processors of the same type, which are on the same switch, all processors cannot be considered to be equidistant in network space. There are two processors per node, and communication between these two processors on the same node is usually faster than inter-node communication. For this reason, we see a difference in runtime between optimal mappings and alternate mappings.

4.2.3 Matrix multiplication on IBM pSeries 655

In the theoretical supply matrix for the 32-processor pSeries machine, there are 8×8 blocks of ones along the diagonal, representing communication between processors on the same node. All other elements are represented by some constant $\beta > 1$. This is because communication cost is theoretically the same between any pair of processors not on the same node, and it is more expensive than intra-node communication. $\beta > 1$ is the same for all inter-node pairs because of the switch nature of the network. The pSeries machine adds an interesting aspect that does not have to be taken into consideration for a machine like the 2^6 QCDOC. Because there is a bus-based connection between processors on a node and at most one communication per given time step may be performed on the bus, we now find possible contention for links in the matrix multiplication problem. The theoretical supply matrix for the 32-processor pSeries machine is displayed in Figure 4.3.

In the experimental supply matrix for the pSeries machine, the entries are

$$c_{i,j} = \begin{cases} 1, & \text{if } i \text{ and } j \text{ are on the same node;} \\ 14.88, & \text{otherwise (different nodes).} \end{cases}$$

The 14.88 represents $\frac{2.50}{.168}$; the conversion is made to give intra-node communication a unit cost.

1D block decomposition: Refer to tables 4.15 to 4.19.

Mapping	Objective Function Value
Optimal	$\frac{7p}{8}(1)(p-1) + \frac{p}{8}(14.88)(p-1) = \frac{p}{8}(p-1)(14.88+7)$
Alternate	$p(p-1)(14.88)$

Table 4.15: Objective function values for 1D block decomposition on pSeries 655

As with Galaxy, the rank order mapping is also the best mapping.

2D block decomposition: Refer to tables 4.20 to 4.24.

Again, the rank order mapping is also the best mapping. The rank order mapping for 2D block decomposition on Galaxy was not the best mapping.

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & \beta & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}$$

Figure 4.3: The theoretical supply matrix for IBM pSeries 655

	Best	**ROMap**	**Alternate**
Objective Function Value	656.4	656.4	3571.2
Normalized Obj. Fc'n Val.	1.00	1.00	5.44
Experimental Comm. Time	0.0594	0.0594	0.512
Normalized ECT	1.00	1.00	8.62
Experimental Total Time	1.64	1.64	1.87
Normalized ETT	1.00	1.00	1.14

Table 4.16: pSeries, N=1024, $P = 16$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	656.4	656.4	3571.2
Normalized Obj. Fc'n Val.	1.00	1.00	5.44
Experimental Comm. Time	0.213	0.213	1.48
Normalized ECT	1.00	1.00	6.95
Experimental Total Time	2.91	12.91	13.55
Normalized ETT	1.00	1.00	1.05

Table 4.17: pSeries, N=2048, $P = 16$, 1D block decomposition

This is due to the fact that Galaxy has two processors per node and the pSeries machine has eight.

4.2.4 Matrix multiplication on Blue Gene/L

For Galaxy and the pSeries machine, it is not possible to assign numerical values to all entries in the theoretical supply matrix based on the description of the machine. Both involve dual-processor nodes and switches; the notion of hop count becomes meaningless. We consequently determined the relative cost of intra-node versus inter-node communication and used the corresponding experimental supply matrix in our analysis. Hop count has meaning and is easy to determine for BG/L and QCDOC, however. Several sizes of BG/L machines at ANL can be studied. The node dimensions of these 3D partitions

	Best	**ROMap**	**Alternate**
Objective Function Value	656.4	656.4	3571.2
Normalized Obj. Fc'n Val.	1.00	1.00	5.44
Experimental Comm. Time	0.451	0.451	3.63
Normalized ECT	1.00	1.00	8.05
Experimental Total Time	49.80	49.80	51.77
Normalized ETT	1.00	1.00	1.04

Table 4.18: pSeries, N=3072, $P = 16$, 1D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	656.4	656.4	3571.2
Normalized Obj. Fc'n Val.	1.00	1.00	5.44
Experimental Comm. Time	0.807	0.807	6.78
Normalized ECT	1.00	1.00	8.40
Experimental Total Time	102.77	102.77	106.03
Normalized ETT	1.00	1.00	1.03

Table 4.19: pSeries, N=4096, $P = 16$, 1D block decomposition

are given in 4.25. Although the machine is commonly described as a 3D torus, it is actually a 3D mesh for partition sizes below 512 [20]. 512 nodes comprise a midplane, and at this point all physical torus connections can be used.

Each processor can be specified in the machine file by its node coordinates in the X, Y, and Z dimensions, with an additional number representing processor number if running in virtual node mode.

For the 32-processor machine, intelligent task mapping only has a slight impact on runtime. This can be attributed to advanced routing techniques. The maximum hop distance between any two nodes in this machines is 7. When we use larger machines, the impact of intelligent task mapping becomes greater.

Mapping	Objective Function Value
Optimal	$p\left(\sqrt{p}-1\right)(1)+\frac{p}{2}\left(\sqrt{p}-1\right)(14.88)+\frac{p}{2}\left(\sqrt{p}-1\right)(1)$
Alternate	$16(2\times 14.88+1+3\times 14.88)$

Table 4.20: Objective function values for 2D block decomposition on pSeries 655

	Best	ROMap	Alternate
Objective Function Value	429.12	429.12	1206.4
Normalized Obj. Fc'n Val.	1.00	1.00	2.81
Experimental Comm. Time	0.0451	0.0451	0.144
Normalized ECT	1.00	1.00	3.19
Experimental Total Time	1.36	1.36	2.23
Normalized ETT	1.00	1.00	1.64

Table 4.21: pSeries, N=1024, $P = 16$, 2D block decomposition

4.2.5 Matrix multiplication on QCDOC

In the 64-processor QCDOC machine, the maximum distance between any two processors is six. This 64-processor QCDOC architecture can also be described as a six-dimensional hypercube.

The theoretical supply matrix for the $2\times 2\times 2\times 2\times 2\times 2$ QCDOC machine is shown in Figure 2. Each entry (i, j) represents the distance between processors i and j on the QCDOC torus. When discussing the machine, each processor can be identified by its coordinates in each of the six dimensions. Two processors are directly connected if they differ by one in a single coordinate only. Since each dimension is of size two in the 64-processor configuration, each processor has one neighbor per dimension. The pairs of processors farthest from each other are those whose coordinates differ by one in all six

	Best	**ROMap**	**Alternate**
Objective Function Value	429.12	429.12	1206.4
Normalized Obj. Fc'n Val.	1.00	1.00	2.81
Experimental Comm. Time	0.156	0.156	0.562
Normalized ECT	1.00	1.00	3.60
Experimental Total Time	10.68	10.68	15.98
Normalized ETT	1.00	1.00	1.50

Table 4.22: pSeries, N=2048, $P = 16$, 2D block decomposition

	Best	**ROMap**	**Alternate**
Objective Function Value	429.12	429.12	1206.4
Normalized Obj. Fc'n Val.	1.00	1.00	2.81
Experimental Comm. Time	0.349	0.349	1.34
Normalized ECT	1.00	1.00	3.84
Experimental Total Time	35.80	35.80	58.02
Normalized ETT	1.00	1.00	1.62

Table 4.23: pSeries, N=3072, $P = 16$, 2D block decomposition

dimensions. Consequently, the maximum distance between any two processors is six. This 64-processor QCDOC architecture may also be described as a six-dimensional hypercube.

Matrix multiplication was implemented on the QCDOC machines at both Columbia and BNL, again using the two basic algorithms described earlier for 1D and 2D block decompositions. The largest Columbia machine readily available consists of 16 processors, while the BNL machine consists of 64. Although they have the same number of processors, a 6D 64-processor ($2 \times 2 \times 2 \times 2 \times 2 \times 2$) machine differs significantly from its 2D (8×8) counterpart. In the first case each processor has six neighbors, but in the second each has only four. However, a 6D 16-processor ($2 \times 2 \times 2 \times 2 \times 1 \times 1$) machine does not differ from its 2D (4×4) counterpart. In both cases each processor has

$$\begin{pmatrix}
0&1&1&2&1&2&2&3&1&2&2&3&2&3&3&4&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&2&3&3&4&3&4&4&5&3&4&4&5&4&5&5&6\\
1&0&2&1&2&1&3&2&2&1&3&2&3&2&4&3&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&3&2&4&3&4&3&5&4&4&3&5&4&5&4&6&5\\
1&2&0&1&2&3&1&2&2&3&1&2&3&4&2&3&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&3&4&2&3&4&5&3&4&4&5&3&4&5&6&4&5\\
2&1&1&0&3&2&2&1&3&2&2&1&4&3&3&2&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&4&3&3&2&5&4&4&3&5&4&4&3&6&5&5&4\\
1&2&2&3&0&1&1&2&2&3&3&4&1&2&2&3&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&3&4&4&5&2&3&3&4&4&5&5&6&3&4&4&5\\
2&1&3&2&1&0&2&1&3&2&4&3&2&1&3&2&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&4&3&5&4&3&2&4&3&5&4&6&5&4&3&5&4\\
2&3&1&2&1&2&0&1&3&4&2&3&2&3&1&2&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&4&5&3&4&3&4&2&3&5&6&4&5&4&5&3&4\\
3&2&2&1&2&1&1&0&4&3&3&2&3&2&2&1&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&5&4&4&3&4&3&3&2&6&5&5&4&5&4&4&3\\
1&2&2&3&2&3&3&4&0&1&1&2&1&2&2&3&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&3&4&4&5&4&5&5&6&2&3&3&4&3&4&4&5\\
2&1&3&2&3&2&4&3&1&0&2&1&2&1&3&2&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&4&3&5&4&5&4&6&5&3&2&4&3&4&3&5&4\\
2&3&1&2&3&4&2&3&1&2&0&1&2&3&1&2&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&4&5&3&4&5&6&4&5&3&4&2&3&4&5&3&4\\
3&2&2&1&4&3&3&2&2&1&1&0&3&2&2&1&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&5&4&4&3&6&5&5&4&4&3&3&2&5&4&4&3\\
2&3&3&4&1&2&2&3&1&2&2&3&0&1&1&2&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&4&5&5&6&3&4&4&5&3&4&4&5&2&3&3&4\\
3&2&4&3&2&1&3&2&2&1&3&2&1&0&2&1&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&5&4&6&5&4&3&5&4&4&3&5&4&3&2&4&3\\
3&4&2&3&2&3&1&2&2&3&1&2&1&2&0&1&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&5&6&4&5&4&5&3&4&4&5&3&4&3&4&2&3\\
4&3&3&2&3&2&2&1&3&2&2&1&2&1&1&0&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&6&5&5&4&5&4&4&3&5&4&4&3&4&3&3&2\\
1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&0&1&1&2&1&2&2&3&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&3&4&4&5&4&5&5&6&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5\\
2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&1&0&2&1&2&1&3&2&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&4&3&5&4&5&4&6&5&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4\\
2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&1&2&0&1&2&3&1&2&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&4&5&3&4&5&6&4&5&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4\\
3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&2&1&1&0&3&2&2&1&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&5&4&4&3&6&5&5&4&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3\\
2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&1&2&2&3&0&1&1&2&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&4&5&5&6&3&4&4&5&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4\\
3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&2&1&3&2&1&0&2&1&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&5&4&6&5&4&3&5&4&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3\\
3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&2&3&1&2&1&2&0&1&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&5&6&4&5&4&5&3&4&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3\\
4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&3&2&2&1&2&1&1&0&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&6&5&5&4&5&4&4&3&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2\\
2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&1&2&2&3&2&3&3&4&0&1&1&2&1&2&2&3&3&4&4&5&4&5&5&6&2&3&3&4&3&4&4&5&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4\\
3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&2&1&3&2&3&2&4&3&1&0&2&1&2&1&3&2&4&3&5&4&5&4&6&5&3&2&4&3&4&3&5&4&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3\\
3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&2&3&1&2&3&4&2&3&1&2&0&1&2&3&1&2&4&5&3&4&5&6&4&5&3&4&2&3&4&5&3&4&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3\\
4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&3&2&2&1&4&3&3&2&2&1&1&0&3&2&2&1&5&4&4&3&6&5&5&4&4&3&3&2&5&4&4&3&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2\\
3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&2&3&3&4&1&2&2&3&1&2&2&3&0&1&1&2&4&5&5&6&3&4&4&5&3&4&4&5&2&3&3&4&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3\\
4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&3&2&4&3&2&1&3&2&2&1&3&2&1&0&2&1&5&4&6&5&4&3&5&4&4&3&5&4&3&2&4&3&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2\\
4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&3&4&2&3&2&3&1&2&2&3&1&2&1&2&0&1&5&6&4&5&4&5&3&4&4&5&3&4&3&4&2&3&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2\\
5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&4&3&3&2&3&2&2&1&3&2&2&1&2&1&1&0&6&5&5&4&5&4&4&3&5&4&4&3&4&3&3&2&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1\\
1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&2&3&3&4&3&4&4&5&3&4&4&5&4&5&5&6&0&1&1&2&1&2&2&3&1&2&2&3&2&3&3&4&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5\\
2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&3&2&4&3&4&3&5&4&4&3&5&4&5&4&6&5&1&0&2&1&2&1&3&2&2&1&3&2&3&2&4&3&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4\\
2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&3&4&2&3&4&5&3&4&4&5&3&4&5&6&4&5&1&2&0&1&2&3&1&2&2&3&1&2&3&4&2&3&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4\\
3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&4&3&3&2&5&4&4&3&5&4&4&3&6&5&5&4&2&1&1&0&3&2&2&1&3&2&2&1&4&3&3&2&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3\\
2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&3&4&4&5&2&3&3&4&4&5&5&6&3&4&4&5&1&2&2&3&0&1&1&2&2&3&3&4&1&2&2&3&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4\\
3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&4&3&5&4&3&2&4&3&5&4&6&5&4&3&5&4&2&1&3&2&1&0&2&1&3&2&4&3&2&1&3&2&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3\\
3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&4&5&3&4&3&4&2&3&5&6&4&5&4&5&3&4&2&3&1&2&1&2&0&1&3&4&2&3&2&3&1&2&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3\\
4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&5&4&4&3&4&3&3&2&6&5&5&4&5&4&4&3&3&2&2&1&2&1&1&0&4&3&3&2&3&2&2&1&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2\\
2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&3&4&4&5&4&5&5&6&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&0&1&1&2&1&2&2&3&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4\\
3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&4&3&5&4&5&4&6&5&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&1&0&2&1&2&1&3&2&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3\\
3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&4&5&3&4&5&6&4&5&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&1&2&0&1&2&3&1&2&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3\\
4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&5&4&4&3&6&5&5&4&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&2&1&1&0&3&2&2&1&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2\\
3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&4&5&5&6&3&4&4&5&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&1&2&2&3&0&1&1&2&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3\\
4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&5&4&6&5&4&3&5&4&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&2&1&3&2&1&0&2&1&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2\\
4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&5&6&4&5&4&5&3&4&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&2&3&1&2&1&2&0&1&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2\\
5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&6&5&5&4&5&4&4&3&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&3&2&2&1&2&1&1&0&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1\\
2&3&3&4&3&4&4&5&3&4&4&5&4&5&5&6&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&0&1&1&2&1&2&2&3&1&2&2&3&2&3&3&4\\
3&2&4&3&4&3&5&4&4&3&5&4&5&4&6&5&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&1&0&2&1&2&1&3&2&2&1&3&2&3&2&4&3\\
3&4&2&3&4&5&3&4&4&5&3&4&5&6&4&5&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&1&2&0&1&2&3&1&2&2&3&1&2&3&4&2&3\\
4&3&3&2&5&4&4&3&5&4&4&3&6&5&5&4&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&2&1&1&0&3&2&2&1&3&2&2&1&4&3&3&2\\
3&4&4&5&2&3&3&4&4&5&5&6&3&4&4&5&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&1&2&2&3&0&1&1&2&2&3&3&4&1&2&2&3\\
4&3&5&4&3&2&4&3&5&4&6&5&4&3&5&4&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&2&1&3&2&1&0&2&1&3&2&4&3&2&1&3&2\\
4&5&3&4&3&4&2&3&5&6&4&5&4&5&3&4&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&2&3&1&2&1&2&0&1&3&4&2&3&2&3&1&2\\
5&4&4&3&4&3&3&2&6&5&5&4&5&4&4&3&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&3&2&2&1&2&1&1&0&4&3&3&2&3&2&2&1\\
3&4&4&5&4&5&5&6&2&3&3&4&3&4&4&5&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&2&3&3&4&3&4&4&5&1&2&2&3&2&3&3&4&1&2&2&3&2&3&3&4&0&1&1&2&1&2&2&3\\
4&3&5&4&5&4&6&5&3&2&4&3&4&3&5&4&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&3&2&4&3&4&3&5&4&2&1&3&2&3&2&4&3&2&1&3&2&3&2&4&3&1&0&2&1&2&1&3&2\\
4&5&3&4&5&6&4&5&3&4&2&3&4&5&3&4&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&3&4&2&3&4&5&3&4&2&3&1&2&3&4&2&3&2&3&1&2&3&4&2&3&1&2&0&1&2&3&1&2\\
5&4&4&3&6&5&5&4&4&3&3&2&5&4&4&3&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&4&3&3&2&5&4&4&3&3&2&2&1&4&3&3&2&3&2&2&1&4&3&3&2&2&1&1&0&3&2&2&1\\
4&5&5&6&3&4&4&5&3&4&4&5&2&3&3&4&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&3&4&4&5&2&3&3&4&2&3&3&4&1&2&2&3&2&3&3&4&1&2&2&3&1&2&2&3&0&1&1&2\\
5&4&6&5&4&3&5&4&4&3&5&4&3&2&4&3&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&4&3&5&4&3&2&4&3&3&2&4&3&2&1&3&2&3&2&4&3&2&1&3&2&2&1&3&2&1&0&2&1\\
5&6&4&5&4&5&3&4&4&5&3&4&3&4&2&3&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&4&5&3&4&3&4&2&3&3&4&2&3&2&3&1&2&3&4&2&3&2&3&1&2&2&3&1&2&1&2&0&1\\
6&5&5&4&5&4&4&3&5&4&4&3&4&3&3&2&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&5&4&4&3&4&3&3&2&4&3&3&2&3&2&2&1&4&3&3&2&3&2&2&1&3&2&2&1&2&1&1&0
\end{pmatrix}$$

Figure 4.4: The theoretical supply matrix for QCDOC

	Best	**ROMap**	**Alternate**
Objective Function Value	429.12	429.12	1206.4
Normalized Obj. Fc'n Val.	1.00	1.00	2.81
Experimental Comm. Time	0.607	0.607	2.29
Normalized ECT	1.00	1.00	3.77
Experimental Total Time	84.70	84.70	124.69
Normalized ETT	1.00	1.00	1.47

Table 4.24: pSeries, N=4096, $P = 16$, 2D block decomposition

Nodes	X	Y	Z
32	4	4	2
64	8	4	2
128	8	4	4
256	8	4	8
512	8	8	8
1024	8	8	16

Table 4.25: BG/L partition dimensions

four neighbors, and the topologies are identical. For these reasons, we focus on the BNL machine for this application because its larger size allows the unique properties of the QCDOC architecture to be exhibited more fully.

Matrix multiplication was first implemented on QCDOC at BNL using the algorithm for a 1D block decomposition described earlier. The A matrix was partitioned into 64 blocks of rows, and the B matrix into 64 blocks of columns. Although six dimensions are available, only one is necessary in order to implement this method of multiplication optimally because of the simple communication pattern. Therefore, the partition was remapped to create a $64 \times 1 \times 1 \times 1 \times 1 \times 1$ ring of processors.

Next, matrix multiplication using the broadcast-multiply-roll (BMR) method for a 2D block decomposition was implemented on the 64-processor machine at BNL. The application matrices A and B were partitioned into 8×8 grids of sub-matrices. Because of the 2D block partition of the application, it is clear that a 2D architecture will produce significantly better results than a 1D (ring) architecture. It should be even more advantageous to utilize the full 6D structure. Timing results were first collected for the simpler case of the 2D $8 \times 8 \times 1 \times 1 \times 1 \times 1$ structure. Although this layout does not take full advantage of the capabilities of the machine, it is easier to envision and possible to make an optimal mapping for by sight. Each of the blocks in the 8×8 decomposition of the matrices can logically be mapped to its corresponding processor in the 8×8 machine matrix.

QMP does not include an appropriate broadcast call; the only available broadcast function is one for the case in which the processor with rank 0 sends data to all other processors. Since our program involves only broadcasts within the rows, we were forced to write our own broadcast by hand. In the first program on the 8×8 structure, broadcast across a row is started by the root and passed from one processor to the next, so only one processor is sending data during each time step. Using this method, the broadcast takes seven steps. In the second program, one processor (the root) sends data during the first time step, and two processors send data simultaneously in each of the next steps until all processors have received the sub-matrix. In this way, the number of time intervals it takes for all processors to receive the matrix is reduced from seven to four. This is also the best we can hope to achieve for the 8×8 configuration. The farthest processor from the broadcasting processor is four hops away, and each processor is only linked directly to two other members of its row, so once it has received and sent once it cannot send again.

By reconfiguring the machine, however, we had eliminated the possibility of using $\frac{1}{3}$ of the communication links. Each processor had only four nearest neighbors out of a potential six, meaning that only 128 links could be utilized; the full 2^6 machine has 192.

After finding the six nearest neighbors of each processor by viewing the processor numbers in terms of their binary representations, a new way of visualizing the 6D network, naturally conducive to matrix multiplication, was discovered. Even though the topology was now that of a 6D torus, the base for the network could still be drawn as an 8×8 grid; this pointed to a clear mapping of blocks to processors. After taking care to create the proper arrangement of processors and adding in four links per processor (to the processors immediately above, below, left, and right), the remaining 64 links were drawn in symmetrically, with four in each row and four in each column. The arrangement of processors is depicted in Figure 4.5 (with each processor identified by the rank assigned to it by the `QMP_get_node_number` call). Suppose r is the processor rank. Note that $\lfloor r/8 \rfloor$ produces the same result for all processors in a row and $r \mod 8$ produces the same result for all processors in a column.

There are 32 communication links in each of the 6 dimensions, yielding a total of 192 links. Using the top row of processors as an example, the links in the first 3 dimensions connect processors in the same row as shown in Figure 4.6 (with the same connection pattern in each row). Similarly, the links in dimensions 3, 4, and 5 connect processors in the same column as shown in

0	4	6	2	3	7	5	1
8	12	14	10	11	15	13	9
24	28	30	26	27	31	29	25
16	20	22	18	19	23	21	17
48	52	54	50	51	55	53	49
56	60	62	58	59	63	61	57
40	44	46	42	43	47	45	41
32	36	38	34	35	39	37	33

Figure 4.5: 2^6 QCDOC processor arrangement

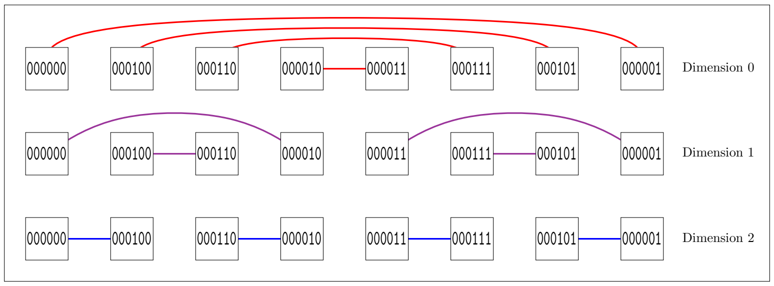

Figure 4.6: QCDOC communication links in the first three dimensions

Figure 4.7 (with the same connection pattern in each column). When all 192 links are drawn in, we have a new way of visually representing the 6D network, well-suited to studying matrix multiplication and other 2D applications, as depicted in Figure 4.8.

The figure points to a way to improve our code. When performing the broadcast of the A sub-matrix within each row using the 2D 8×8 grid, the lower bound on the number of time intervals it takes for all processors to receive the sub-matrix is four. When the full 2^6 grid is used, however, each row has an additional four links, each one connecting two processors that are three hops apart in the 8×8 grid. This enables all processors in a row to receive the A sub-matrix after only three ($\log_2 \sqrt{64}$) time intervals, because

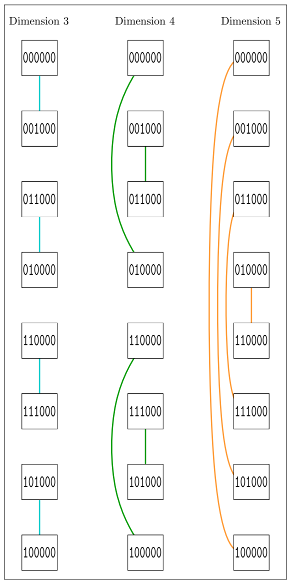

Figure 4.7: QCDOC communication links in the last three dimensions

processors can send more than once after receiving. The first send occurs along the new link out of the broadcasting processor. At this point, two processors have the data. Each sends to its neighbor in the direction away from the link it received on. Now, four processors have the data. The difference at this point is that each of these four processors still has a link to send on during the next time interval, allowing the remaining four processors to receive the data. (In the 8×8 setup, only the two processors that received during the previous time interval still had links to send on.) We cannot possibly improve upon this because there are still processors three hops away from each other in each row; therefore, the mapping is optimal.

We compare runtimes for the matrix multiplication code using the optimal mapping on the three network setups described above. Although this does not deal with task mapping directly, it is an important issue to examine.

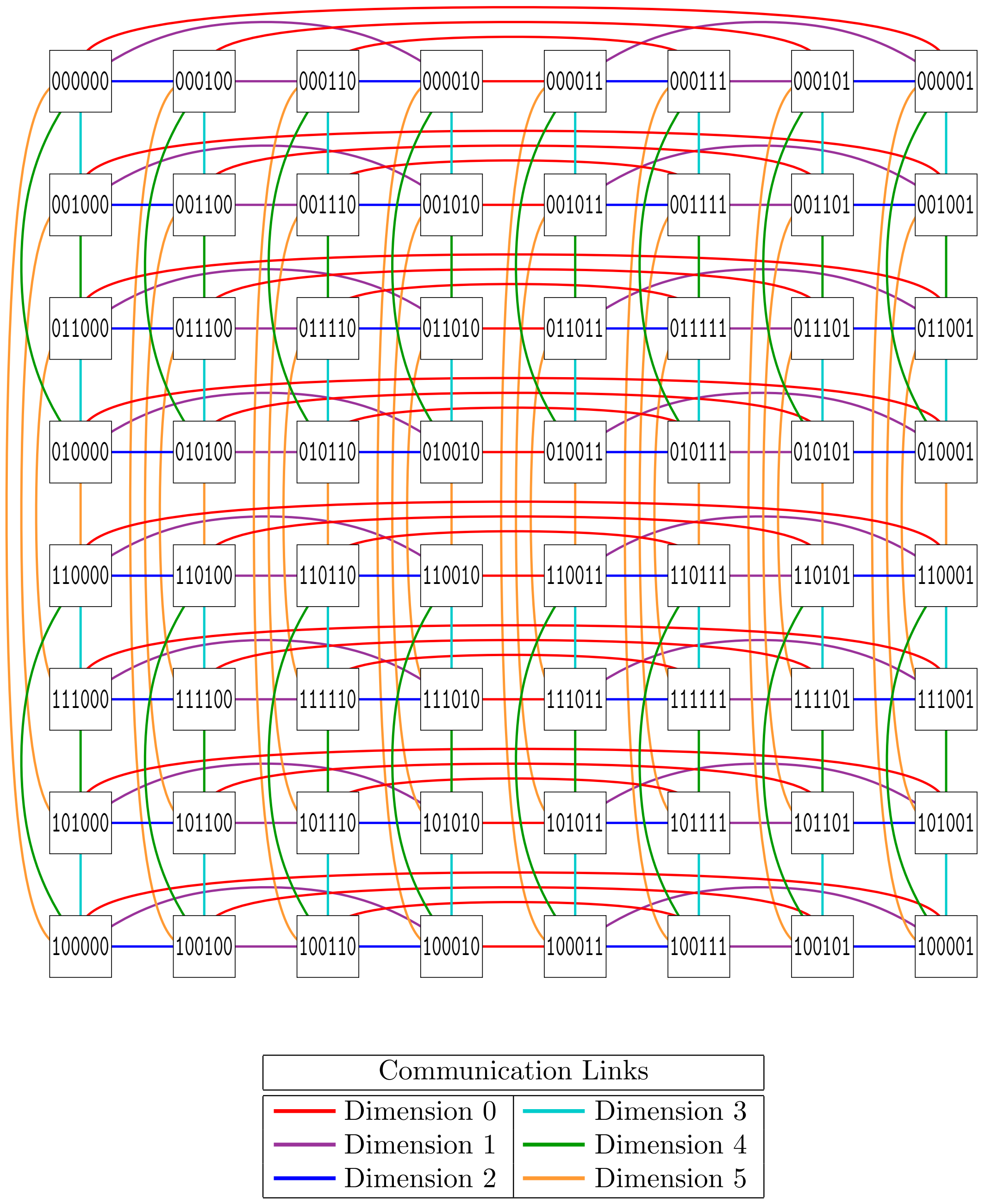

Figure 4.8: The full 2^6 QCDOC processor layout and communication links

	`block1`	`block2`	
Size	Time (s)	Time (s)	Improvement over `block1`
1024	0.105	0.065	38%
2048	0.400	0.244	39%
3072	0.894	0.545	39%

		`block3`	
Size	Time (s)	Improvement over `block1`	Improvement over `block2`
1024	0.049	53%	25%
2048	0.181	55%	26%
3072	0.399	55%	27%

Table 4.26: Comparison of runtimes on three network setups

Our findings illustrate the significance of the connection method in a network of processors. Adding more wires to a network clearly increases its cost, but we show that it also enables problems to be solved more quickly. In future work it may be interesting to look at this tradeoff between cost and runtime improvement. We summarize our results in Table 4.26.

QCDOC differs significantly from the other three machines we examine in that it does not support non-nearest neighbor communication. When we wish to send a message from one processor to another, the exact message path must be specified in the code. On the other three machines, only the sending and receiving processors must make function calls in the code. MPI handles the path the message takes. With QCDOC, however, the programmer must decide upon a path before execution, and each intermediate processor along the path must make the appropriate function call(s). The complexity of the code, therefore, increases as the mapping quality decreases. For this reason, comparing runtimes for various mappings becomes extremely inefficient. Until automatic parallelization becomes feasible, we choose to compare only theoretical results for QCDOC. We assert that task mapping is important for this machine, but code should only be written for the mapping determined to be best. If our model produces positive experimental results for the other three machines, we assume that it will be similarly beneficial for QCDOC.

Because we have already proven an optimal mapping for QCDOC, the conclusion of our study of this problem on this supercomputer is slightly different from what the standard procedure will be in future work. In terms of the cost equation, the mapping we describe has a cost $C = 1216$. Since we have proven that this is the optimal mapping, we want to observe whether the objective function above also gives 1216 as its best cost. We begin with an initial mapping of task i to processor i, for $i = 0, 1, \ldots, 63$. The cost of this mapping is 1552. We then apply simulated annealing to minimize the objective function value. The current simulated annealing method finds a cost of 1216 after approximately 10 seconds. Even after running it substantially longer, this is the lowest cost achieved. However, for different runs, different mappings are associated with this cost.

We would like to examine larger, less symmetric, QCDOC architectures of the form $2 \times 2 \times 2 \times K \times M \times N$ in order to demonstrate the necessity of a model for matrix multiplication. We plan to obtain theoretical results for partitions consisting of 512 ($K = M = N = 4$), 1024 ($K = M = 4$, $N = 8$), 2048 ($K = M = 4$, $N = 16$), and 4096 ($K = M = N = 8$) processors. Contention for links in the 64-processor machine is not an issue, so we can

P	K M N
64	2×2×2× 2 × 2 × 2
512	2×2×2× 4 × 4 × 4
1024	2×2×2× 4 × 4 × 8
2048	2×2×2× 4 × 4 × 16
4096	2×2×2× 8 × 8 × 8

Table 4.27: Larger QCDOC partition dimensions

avoid it in a model for this smaller architecture; for a larger, less symmetric machine, however, it may come into play.

	Best	**ROMap**	**Alternate**
Objective Function Value	1216.00	1552.00	3052.00
Normalized Obj. Fc'n Val.	1.00	1.28	2.51

Table 4.28: QCDOC, P=64, 2D block decomposition

4.3 Arbitrary demand matrices: theoretical and experimental results

We look to related work [8, 11, 27, 55] to find communication patterns common to large classes of applications. Constructing random demand matrices will lead to interesting theoretical results. However, because our primary goal is to contribute to task mapping knowledge for real applications, we currently focus on demand matrices comparable to ones we would expect to find in a wide variety of actual problems. We will address random demand matrices in future work.

4.3.1 Nearest-neighbor communication

For many parallel applications, the demand matrix is sparse [11]. Tasks communicate mostly with nearby tasks. For this reason, it is important to test our models on this type of communication pattern.

In 2D nearest-neighbor communication, each task must send messages to its four neighbors in a 2D grid. Related work [8, 55] has recognized this as a pattern common to many applications, including Jacobi-like applications.

In 3D nearest-neighbor communication, each task must send messages to its six neighbors in a 3D mesh. Related work [11] has recognized this as a pattern common to many applications.

There are also variants of these nearest-neighbor communication patterns, in which each task sends full-sized messages to its nearest neighbors and smaller messages to its next-nearest neighbors in the same direction [11], or each task sends some number of messages to its nearest neighbors and some fraction of that number of messages to random tasks farther away [55], for both 2D [55] and 3D problems [11].

Our initial plan was to mimic this work for 2D and 3D meshes and tori. After further thought, however, the decision was made to modify this plan. Many applications in the Harwell-Boeing test collection of matrices [25, 26] exhibit various forms of nearest-neighbor or next-nearest neighbor communication. Studying nearest-neighbor communication in real applications provides more than simply interesting theoretical results. In addition, we choose not to focus on 3D applications because one of the four machines we study, BG/L, is 3D in nature. There should be a natural mapping of 3D applications to this architecture, so intelligent task mapping should not be as necessary for efficiency.

4.3.2 Assorted applications

We use matrices in the Harwell-Boeing test collection [25, 26] to serve as example demand matrices with a wide variety of communication patterns.

In the applications discussed previously, each processor communicates in the same way with its nearest and next-nearest neighbors. Not all real problems exhibit this characteristic. For this reason, we choose to examine some applications from the collection with more irregular communication patterns, as well as some with regular communication patterns.

We test the following 13 applications from the collection:

- Power network patterns (BCSPWR01 and BCSPWR02)
- Finite element structures problems in aircraft design (CAN24, CAN61, and CAN62)
- Small, simple problems (JGL009, JGL011, and RGG010)
- Structural engineering (DWT59)
- Laplace finite element applications (LAP25)
- IBM conference advertisement (IBM32)
- Biochemical ordinary differential equations (CURTIS54)
- Jacobian of emitter-follower-current switch circuit (WILL57)

In some cases the demand matrices were modified slightly to accommodate machine size. Not all applications were tested on all sizes of all architectures, as doing so would require significant modifications and would destroy the true character of the demand matrix.

All row and column headings are the same as those for matrix multiplication except for one. The column heading 'Alternate' has been replaced with 'Worst.' This difference is due to the way in which this mapping is obtained. For matrix multiplication, a poor mapping was chosen by hand. For the following applications, however, we let simulated annealing find the largest, or worst, objective function value it can. This is in contrast to finding the smallest, or best, objective function value.

Galaxy

We examine JGL009, JGL011, and RGG010 for 16 processors.

	Best	**ROMap**	**Worst**
Objective Function Value	101.84	107.12	115.92
Normalized Obj. Fc'n Val.	1.00	1.05	1.14
Experimental Comm. Time	0.852	0.938	1.03
Normalized ECT	1.00	1.10	1.21

Table 4.29: Galaxy, P=16, JGL009

	Best	**ROMap**	**Worst**
Objective Function Value	166.32	169.84	178.64
Normalized Obj. Fc'n Val.	1.00	1.02	1.07
Experimental Comm. Time	1.25	1.27	1.35
Normalized ECT	1.00	1.02	1.08

Table 4.30: Galaxy, P=16, JGL011

Mapping the tasks in rank order is near-optimal for JGL011 on 16 processors. The objective function value for the rank order mapping for is not as good as the value for the best mapping found by the model, but it is very close to it. In the mapping determined by the model, only 3 of the 16 tasks are not mapped to processors with corresponding rank.

Note, in Table 4.30, that the normalized experimental times are almost identical to the normalized objective function values for this instance.

	Best	**ROMap**	**Worst**
Objective Function Value	164.56	168.08	178.64
Normalized Obj. Fc'n Val.	1.00	1.02	1.09
Experimental Comm. Time	1.29	1.31	1.38
Normalized ECT	1.00	1.02	1.07

Table 4.31: Galaxy, P=16, RGG010

For RGG010 on 16 processors we have a situation similar to that observed in Table 4.30 for JGL011. We can see in Table 4.31 that the relative theoretical values predict the relative experimental values very well.

We also observe only a small difference between the best and worst mappings for the three applications above. This can be mainly attributed to the fact that the demand matrices for these applications are fairly dense and the communication volume is identical for all communicating pairs of processors. 61.7%, 62.8%, and 76% of the demand matrix entries are nonzero, for the three applications, respectively.

We examine CAN24, LAP25, and IBM32 for 24 processors.

Although our task mapping model predicts some improvement in runtime for CAN24 on 24 processors, the experiments show that it does not have a

	Best	**ROMap**	**Worst**
Objective Function Value	163.80	179.64	184.92
Normalized Obj. Fc'n Val.	1.00	1.10	1.13
Experimental Comm. Time	0.852	0.857	0.868
Normalized ECT	1.00	1.01	1.02

Table 4.32: Galaxy, P=24, CAN24

significant impact (see Table 4.32.

	Best	**ROMap**	**Worst**
Objective Function Value	320.04	340.16	374.60
Normalized Obj. Fc'n Val.	1.00	1.06	1.17
Experimental Comm. Time	1.57	1.86	2.00
Normalized ECT	1.00	1.18	1.27

Table 4.33: Galaxy, P=24, LAP25

We see in Table 4.33 that the experimental improvement was greater than the theoretical improvement in both cases for LAP25 on 24 processors.

	Best	**ROMap**	**Worst**
Objective Function Value	141.64	176.84	192.20
Normalized Obj. Fc'n Val.	1.00	1.25	1.39
Experimental Comm. Time	0.973	1.05	1.11
Normalized ECT	1.00	1.08	1.14

Table 4.34: Galaxy, P=24, IBM32

We see the opposite in Table 4.34 for IBM32. The theoretical improvement was greater than the experimental improvement.

We examine the differences in these two applications to determine a reason for the difference in theoretical and experimental results. LAP25 has a more regular communication pattern and more nonzero matrix entries than IBM32. LAP25 tasks also communicate mostly with closely neighboring tasks, while IBM32 tasks do not. The difference that affects our observation the most, however, is the symmetry in LAP25 that does not exist in IBM32.

Because LAP25 is symmetric, if the processor to which task i is assigned sends to the processor to which task j is assigned, communication also occurs in the other direction. On a machine like Galaxy, the only differentiating factor for network distance is whether two processors are on the same or different nodes. With a machine like Galaxy, if one communication is made intra-nodal, another one will is also guaranteed to be intra-nodal.

The other factors above also affect the observation. CAN24 is also symmetric, but it has a more irregular communication pattern.

We examine BCSPWR01, CAN24, LAP25, and IBM32 for 32 processors.

	Best	**ROMap**	**Worst**
Objective Function Value	67.92	80.24	104.88
Normalized Obj. Fc'n Val.	1.00	1.18	1.54
Experimental Comm. Time	0.493	0.547	0.579
Normalized ECT	1.00	1.11	1.17

Table 4.35: Galaxy, P=32, BCSPWR01

	Best	**ROMap**	**Worst**
Objective Function Value	163.80	179.64	184.92
Normalized Obj. Fc'n Val.	1.00	1.10	1.13
Experimental Comm. Time	0.965	0.993	1.03
Normalized ECT	1.00	1.03	1.07

Table 4.36: Galaxy, P=32, CAN24

We examine BCSPWR02, CAN61, CAN62, DWT59, CURTIS54, and WILL57 for 64 processors.

On 64 processors, BCSPWR02 is the application with the most significant

	Best	**ROMap**	**Worst**
Objective Function Value	355.20	362.24	397.44
Normalized Obj. Fc'n Val.	1.00	1.02	1.12
Experimental Comm. Time	1.61	1.66	1.74
Normalized ECT	1.00	1.03	1.08

Table 4.37: Galaxy, P=32, LAP25

	Best	**ROMap**	**Worst**
Objective Function Value	224.24	252.40	259.44
Normalized Obj. Fc'n Val.	1.00	1.13	1.16
Experimental Comm. Time	1.11	1.14	1.25
Normalized ECT	1.00	1.03	1.13

Table 4.38: Galaxy, P=32, IBM32

difference among mappings, as seen in Table 4.39. BCSPWR01 displays the most significant difference for 32 processors, as seen in Table 4.35.

As a whole, we do not observe dramatic differences among the various mappings for Galaxy. This can be attributed to the relative homogeneity of the machine. Nonzero entries in the supply matrix take on one of only two nonzero values, and these two values are not drastically different.

IBM pSeries 655

We examine JGL009, JGL011, and RGG010 for 16 processors.

We observe that the difference among mappings is much greater in all cases for the pSeries machine than for Galaxy. This can be attributed to the greater difference between intra-node and inter-node communication time on the pSeries.

We examine CAN24, LAP25, and IBM32 for 24 processors.

We examine BCSPWR01, CAN24, LAP25, and IBM32 for 32 processors.

	Best	**ROMap**	**Worst**
Objective Function Value	39.36	53.44	55.20
Normalized Obj. Fc'n Val.	1.00	1.36	1.40

Table 4.39: Galaxy, P=64, BCSPWR02

	Best	**ROMap**	**Worst**
Objective Function Value	631.68	656.32	684.48
Normalized Obj. Fc'n Val.	1.00	1.04	1.08

Table 4.40: Galaxy, P=64, CAN61

As with Galaxy, the power network application shows the most significant difference among mappings, as seen in Table 4.51.

4.4 Summary of case studies

The results above provide us with many insights into task mapping as a whole. All experimental values given are the average runtimes over hundreds of runs. Galaxy and the pSeries machine exhibit more variance in runtime because other users are sometimes simultaneously accessing the same partitions or connections. BG/L and QCDOC exhibit less variance among runs.

As seen above, when communication occurs among a large percentage of processor pairs, and this communication does not vary significantly in volume, we do not observe a significant improvement when using a mapping determined by the model instead of an arbitrary one.

We should also highlight the similarities and differences between Galaxy and the pSeries 655. For both machines, each node contains two processors. Additionally, nonzero entries in the supply matrices take on only one of two values. The significance of task mapping is not the same on both machines,

	Best	**ROMap**	**Worst**
Objective Function Value	164.24	187.12	215.28
Normalized Obj. Fc'n Val.	1.00	1.14	1.31

Table 4.41: Galaxy, P=64, CAN62

	Best	**ROMap**	**Worst**
Objective Function Value	236.00	250.08	287.04
Normalized Obj. Fc'n Val.	1.00	1.06	1.22

Table 4.42: Galaxy, P=64, DWT59

however. More improvement can be made through mapping on the pSeries. A difference that brings about this inconsistency is the fact that inter-node communication for the pSeries machine is very expensive in comparison to intra-node communication.

Through QCDOC, we learn that task mapping analysis is difficult for machines for which non-nearest neighbor communication has not been implemented. It is easy to compare theoretical runtimes for different mappings, but it is inefficient to implement code for these mappings. Because non-nearest neighbor communication is not supported, we cannot simply use an alternative machine file. Current code must be modified drastically. Task mapping is still useful if we only wish to determine the best mapping and implement it. A better mapping leads to fewer lines of code because communications are more efficient; message path lengths are shorter.

General observations are discussed further in Chapter 6.

	Best	**ROMap**	**Worst**
Objective Function Value	559.08	592.52	654.12
Normalized Obj. Fc'n Val.	1.00	1.06	1.17

Table 4.43: Galaxy, P=64, CURTIS54

	Best	**ROMap**	**Worst**
Objective Function Value	521.44	544.32	618.24
Normalized Obj. Fc'n Val.	1.00	1.04	1.19

Table 4.44: Galaxy, P=64, WILL57

	Best	**ROMap**	**Worst**
Objective Function Value	139.16	208.56	416.76
Normalized Obj. Fc'n Val.	1.00	1.50	2.99
Experimental Comm. Time	0.422	0.640	1.05
Normalized ECT	1.00	1.52	2.49

Table 4.45: pSeries, P=16, JGL009

	Best	**ROMap**	**Worst**
Objective Function Value	371.36	537.92	635.08
Normalized Obj. Fc'n Val.	1.00	1.45	1.71
Experimental Comm. Time	0.979	1.06	1.38
Normalized ECT	1.00	1.08	1.41

Table 4.46: pSeries, P=16, JGL011

	Best	**ROMap**	**Worst**
Objective Function Value	315.84	413.00	607.32
Normalized Obj. Fc'n Val.	1.00	1.31	1.92
Experimental Comm. Time	0.728	0.967	1.27
Normalized ECT	1.00	1.33	1.74

Table 4.47: pSeries, P=16, RGG010

	Best	**ROMap**	**Worst**
Objective Function Value	344.6	747.12	899.80
Normalized Obj. Fc'n Val.	1.00	2.17	2.61

Table 4.48: pSeries, P=24, CAN24

	Best	**ROMap**	**Worst**
Objective Function Value	636.68	954.92	1760.96
Normalized Obj. Fc'n Val.	1.00	1.50	2.77

Table 4.49: pSeries, P=24, LAP25

	Best	**ROMap**	**Worst**
Objective Function Value	336.72	683.72	975.20
Normalized Obj. Fc'n Val.	1.00	2.03	2.90

Table 4.50: pSeries, P=24, IBM32

	Best	**ROMap**	**Worst**
Objective Function Value	65.76	232.32	565.44
Normalized Obj. Fc'n Val.	1.00	3.53	8.60

Table 4.51: pSeries, P=32, BCSPWR01

	Best	**ROMap**	**Worst**
Objective Function Value	344.60	747.12	941.44
Normalized Obj. Fc'n Val.	1.00	2.17	2.73

Table 4.52: pSeries, P=32, CAN24

	Best	**ROMap**	**Worst**
Objective Function Value	810.24	1060.08	2031.68
Normalized Obj. Fc'n Val.	1.00	1.31	2.51

Table 4.53: pSeries, P=32, LAP25

	Best	**ROMap**	**Worst**
Objective Function Value	690.84	1037.84	1329.32
Normalized Obj. Fc'n Val.	1.00	1.50	1.92

Table 4.54: pSeries, P=32, IBM32

Chapter 5

Automatic Parallelization: Thoughts and Practices

As stated earlier, the goal for the future is automatic parallelization, or the conversion of sequential code to parallel code by a compiler. In practice today, this translation is normally done by hand.

Automatic parallelization methods used today have many imperfections. They often lose their efficiency quickly when the number of processors extends beyond a relatively small number. Because of a desire for fast compilation, accuracy is sometimes lost [2].

Researchers at the University of Tennessee are attempting to make advances in this area with their SANS (Self-Adapting Numerical Software) systems [13]. One of the key facets of this proposed software is its optimized communication library, which will optimize MPI operations based on properties such as network topology and message size.

5.1 Semi-automatic parallelization

As its name suggests, semi-automatic parallelization is a combination of manual and automatic parallelization. A semi-automatic parallelization tool takes sequential code and a parameterized description of the machine architecture as input. Its output consists of both comments in the original program indicating places where parallelization may be possible and new suggested parallel code taking advantage of those possibilities. Additional information is provided, enabling the user to understand what is being suggested in order to further improve the parallel code. The parallel code produced by the tool alone is almost certainly suboptimal, but allowing active human participation in the parallelization process leads to favorable possibilities for improvement.

Many attempts at semi-automatic parallelization have been made over the years. The PARTITA system was designed in 1994 to aid in the parallelization of FORTRAN programs [39]. Parallel Software Products Inc. has been working on a similar toolkit for over 15 years, with its most recent product, ParaWise 2.4, being released in 2004 [2]. ParaWise boasts of a friendly, yet detailed, user interface that significantly aids in producing efficient code. Software packages such as these two propose to be cost-effective solutions to parallelizing applications.

Chapter 6

Summary and Conclusions

Because task mapping is an application- and machine-specific problem, the standard approach in previous work has been to examine a particular problem on a single parallel computer. Although surveys giving descriptions of multiple instances have been compiled, they have been summaries of distinct findings rather than efforts to draw more general conclusions. Difficulties are encountered when deciding to test a new application on the same machine or to run the same application on a new machine. A model which was found to produce an optimal mapping for one type of input may be extremely poor for another because of differences in the topology of the networks or the communication pattern of the problem [10]. Our work seeks to patch together a collection of specific problems so that we have a sizeable enough picture of task mapping that we can step back and examine the field in a more general way. As we become familiar with even more applications on more machines, we can compare patterns of supply and demand matrices and identify characteristics by which to classify them. We can similarly examine patterns in objective functions. By analyzing which models work best for which types of problems and supercomputers, we can move toward our ultimate objective of automatic parallelization. We will also be producing nontrivial results for specific instances as we approach this final goal.

Our work is the first to perform a detailed theoretical and experimental analysis on four physical machines of such varying architectures. In addition, we have studied a wide variety of application types on these computers. For this reason, we believe that we are in a position to make educated statements about task mapping in general.

This is also the first study to address task mapping on an actual 6D architecture. Previous work has only simulated such a network [24].

Another advantage of our work over other previous work is our use of actual rather than simulated parallel systems. When simulating networks, many assumptions regarding aspects such as link transfer time and node delay [24] are made. These assumptions may be proved false when an attempt is made to implement the mapping strategy on an actual system.

Based on our work with matrix multiplication and the applications from the Harwell-Boeing collection, we currently have a small piece of the puzzle in our hands. There is essentially no limit to the number of applications that can be studied. Even the number of types of supercomputers that can be utilized may be thought of as limitless, because new designs can lead to the construction of new machines [64]. We certainly do not make the claim that we can study all possible combinations. We are confident, however, that we will be able to set forth general guidelines based on the conclusions that we draw from the specific instances that we do study. These, in turn, can be applied to the occurrences that we have not examined after some characterizations describing them have been made.

Although most of our experimental results are on relatively small numbers of processors, we are confident that we will have similar success with massively parallel machines. Most previous work has also considered systems of 64 [24] or fewer processors. When massively parallel supercomputers are studied, the typical procedure is to focus on no more than a few applications on one machine. Results have proven the continued importance of task mapping for such larger machines.

6.1 Characterizing architectures

There are a number of factors by which architectures can be characterized, including the following:

- type (Beowulf, SMP, cellular, etc.)
- number of distinct values in supply matrix
- diameter [8]
- average inter-processor distance [8]
- number of processors [10]
- number of communication links [10]
- average number of links per processor [10]

- variance in computing capabilities

The number of distinct values in the supply matrix can affect the significance of task mapping. Supply matrices for machines like Galaxy and the pSeries 655 have only two distinct nonzero values. The supply matrix for QC-DOC has six distinct nonzero entries. In general, more distinct values lead to more improvement through task mapping. However, the difference between these values is also a factor. If a machine has many different inter-node distances, but they are all very close in value, the order in which the tasks are mapped is almost inconsequential.

Whether or not all processors have equal computing capabilities is also a characteristic by which architectures can be differentiated. It should affect relevance of task mapping. However, because we have not yet studied heterogeneous architectures, we cannot discuss its effect here.

6.2 Characterizing applications

We can similarly characterize applications. We can consider aspects such as the following:

- number of times the application will be run
- total number of messages [47]
- number of sources [47]
- number of destinations [47]

- distribution of message size [47]
- distribution of pairwise number of sends
- type of communication (one/some/all to one/some/all)
- density of demand matrix
- symmetry of demand matrix
- communication-to-computation ratio [22, 24]

The number of times the application will be run should affect the choice of heuristic. If the application is to be run only once, using a heuristic with greater time complexity may not be efficient. If it is to be run many times, however, the benefit of such a heuristic may outweigh the time cost.

All communications may be viewed as coming from one, some, or all processors and going to one, some, or all processors. There are general MPI send calls associated with each of the nine possible combinations. In an application like matrix multiplication using the broadcast-multiply-roll method, the broadcast step involves $\sqrt{n}$ one-to-some communications and the roll step involves n one-to-one communications. Profiling tools can produce this information.

We observe in 4.3.2 that the density or sparsity of a demand matrix can affect the significance of intelligent task mapping. We can express this quantity as

$$\text{density} = \frac{\text{number of pairs } (i, j) \text{ for which } t_i \text{ communicates to } t_j}{t^2}. \tag{6.1}$$

This is the same as the fraction of nonzero entries in the demand matrix.

If this quantity is large, it may be a contributing factor to the decision that the time spent to determine a mapping from a model is too costly for the expected improvement in runtime.

The communication-to-computation ratio may also affect the decision on whether to implement a task mapping model. However, since we have only studied applications for which perfect load balance is easily achievable, we cannot draw conclusions in this area at present.

6.3 Future work

Although much has been discovered, we are still left with many promising avenues to pursue. Because of the great number of applications, objective functions, heuristics, problem sizes, methods of creating supply and demand matrices, and even parallel computers themselves, we are unable to examine every possible combination in this work. In some cases, we have made certain assumptions to narrow our focus. We will explore what happens when we remove some of them. In other cases, we have moved on to a new problem before thoroughly exhausting all ideas about the current problem in order to gain a better perspective on task mapping as a whole. We can revisit these problems.

Thus far, we have assumed that the code for the applications we study will be run many times so the time to solution for finding a mapping is negligible in comparison to the sum of runtimes for all runs of the code. For large problems whose code will only be run once, however, some methods of task mapping may be relatively costly with respect to time. For this reason, we will consider time to solution for finding a mapping in addition to time to solution for the application itself [24]. We may wish to look at the ratios of these two times. This will involve a thorough analysis of computational complexity of the various mapping strategies.

Along the same lines, we will consider ideas presented in [24] and construct a new heuristic algorithm to use in place of simulated annealing to optimize a given objective function. When using a demand matrix to represent an application, it is easy to identify tasks that communicate heavily and those that do not. A simple comparison can be made by summing the entries for each row in the matrix. We may want to develop a heuristic that focuses mainly on where heavily-communicating tasks get mapped and spends less computational time considering lightly-communicating tasks. This strategy could result in a decrease in the time taken to find an efficient mapping, which would be especially significant for large applications and large architectures.

We will also examine alternative metrics for cost of communication in the supply matrix. Bhanot et al. [11] propose, but never evaluate, two distance metrics as alternatives to the hop count for BG/L. The first is

$$d(a,b) = \frac{3}{D} h(a,b), \tag{6.2}$$

where D is the number of torus dimensions used in the shortest path from a to b. The second is the Euclidean distance between processors,

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}. \tag{6.3}$$

We will study the effect of using these metrics on BG/L and QCDOC and compare the results to those obtained using the hop count. We will also examine use of the virtual node mode for BG/L.

Additionally, we plan to study more matrices from the Harwell-Boeing collection, incorporate message size into the demand matrix, and examine heterogeneous systems with processors of differing computational power.

Other possibilities for study are problems that have been decomposed into a larger number of tasks than there are processors. In these cases we will consider load balancing. We may also drop the assumption that there is no ordering on the tasks. In this case we will look at task precedence graphs. Thus far we have only studied static mapping; we may also consider dynamic mapping.

This work need not be restricted to the ideas of scientific computing; it may also reach out to business computing. An example of previous success can be found in Songnian Zhou's Platform Computing, a company employing grid computing software technology, in particular LSF (Load Sharing Facility) software, to increase the business performance of over 1,700 organizations worldwide [4]. The abstract for a patent of Sheets et al. for a "method and system for providing dynamic hosted service management across disparate accounts/sites" [61] also contains promising implications for task mapping. By examining both scientific and business applications, we give ourselves a wide variety of areas in which to make advances.

Bibliography

[1] Galaxy, 2004. Department of Applied Mathematics and Statistics, Stony Brook University, `http://galaxy.ams.sunysb.edu`.

[2] Parawise: The computer aided parallelization toolkit, 2004. Parallel Software Products Inc., `http://www.parallelsp.com/parawise.htm`.

[3] Lattice QCD message passing (QMP), 2005. The Lattice Web: A Resource for the International Lattice Gauge Theory Community, `http://www.lqcd.org/QMP`.

[4] Platform Computing - Accelerating Intelligence - Grid Computing, 2005. Platform Computing, `http://www.platform.com`.

[5] M. Affenzeller and R. Mayrhofer. Generic heuristics for combinatorial optimization problems. In *Proceedings of 9th International Conference on Operational Research (KOI)*, pages 83–92, 2002.

[6] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[7] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development*, 38:673–681, 1994.

[8] T. Agarwal, A. Sharma, and L. V. Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[9] J. Aguilar and E. Gelenbe. Task assignment and transaction clustering heuristics for distributed systems. *Information Sciences*, 97(1/2):199–219, 1997.

[10] T. Baba, Y. Iwamoto, and T. Yoshinaga. A network-topology independent task allocation strategy for parallel computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 878–887, Washington, DC, 1990. IEEE Computer Society.

[11] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, and R. Walkup. Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):489–500, 2005.

[12] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[13] G. Bosilca. Self adapting numerical software SANS effort. *Submitted to IBM Journal of Research and Development*, 2005.

[14] D. C. Bossen, A. Kitamorn, K. F. Reick, and M. S. Floyd. Fault-tolerant design of the IBM pSeries 690 system using POWER4 processor technology. *IBM Journal of Research and Development*, 46(1):77–86, 2002.

[15] T. Bultan and C. Aykanat. A new mapping heuristic based on mean field annealing. *Journal of Parallel and Distributed Computing*, 16(4):292–305, 1992.

[16] L. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State University, 1969.

[17] J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. *Concurrency: Practice and Experience*, 10(8):655–670, 1998.

[18] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[19] N. H. Christ. QCDOC project hardware status, 2004. 2004 Brookhaven National Laboratory All-Hands Meeting, http://www.bnl.gov/lqcd/comp/qcdoc_bkgnd.asp.

[20] S. Coghlan. Introduction to BG/L: Argonne leadership computing facility, February 2007.

[21] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proceedings of 46th Annual Symposium on Foundations of Computer Science*, pages 379–388, Pittsburgh, PA, 2005.

[22] J. B. Collins. An approach to scheduling task graphs with contention in communication. 2001.

[23] Y. Deng, J. Glimm, and J. W. Davenport. Global communication schemes on QCDOC. *Submitted to IEEE Transactions on Parallel and Distributed Computing*, 2003.

[24] V. A. Dixit-Radiya and D. K. Panda. Task assignment on distributed-memory systems with adaptive wormhole routing. In *Proceedings of Symposium on Parallel and Distributed Processing*, pages 674–681, 1993.

[25] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989.

[26] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Chilton, Oxon, England, 1992.

[27] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain. Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements. *IBM Journal of Research and Development*, 49(2/3):457–464, 2005.

[28] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, 10(1):35–44, 1990.

[29] A. Gara et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[30] P. A. Boyle et al. Status of the QCDOC project. *Nuclear Physics Proceedings Supplements*, 106:177–183, 2002.

[31] P. A. Boyle et al. QCDOC: a 10 teraflops computer for tightly-coupled calculations. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2004.

[32] P. A. Boyle et al. Overview of the QCDSP and QCDOC computers. *IBM Journal of Research and Development*, 49(2/3):351–365, 2005.

[33] B. Fox. Integrating and accelerating Tabu search, simulated annealing and genetic algorithms. *Annals of Operations Research*, 41:47–67, 1993.

[34] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube I: matrix multiplication. *Parallel Computing*, 3:17–31, 1987.

[35] B. Grayson, A. P. Shah, and R. A. van de Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[36] A. Gupta and V. Kumar. Scalability of parallel algorithms for matrix multiplication. In *Proceeedings of 1993 International Conference on Parallel Processing*, volume III, pages 115–123. CRC Press, 1993.

[37] I. Haritaoglu and C. Aykanat. An efficient mapping heuristic for mesh-connected parallel architectures based on mean field annealing. *Lecture Notes in Computer Science*, 854:820–831, 1994.

[38] W. E. Hart. Tabu search, 1997. Sandia National Laboratories, `http://www.cs.sandia.gov/opt/survey/ts.html`.

[39] L. Hascoet. PARTITA parallelization and code generation. *EUREKA Project 933 EUROTOPS*, 1994.

[40] H.-U. Heiss and M. Dormanns. Mapping tasks to processors with the aid of Kohonen networks. In *Proceedings of High Performance Computing Conference '94*, pages 133–143, Singapore, 1994.

[41] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[42] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, 6(7):571–594, 1994.

[43] J. Khoriaty. Kernel performance on QCDOC. Master's thesis, The University of Edinburgh, 2005.

[44] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[45] M. Krishnan and J. Nieplocha. SRUMMA: A matrix multiplication algorithm suitable for clusters. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004.

[46] A. Kumar. Task layout optimizer for Blue Gene, 2005. IBM, `http://www.alphaworks.ibm.com/tech/bglmap`.

[47] S.-Y. Lee. Effects of communication characteristics on task mapping quality on a 2-d mesh with wormhole routing, 2000.

[48] E. J. Lerner. Cellular architecture builds next generation supercomputers, 2001. IBM, `http://www.research.ibm.com/thinkresearch/pages/2001/20010611_cellular.shtml`.

[49] T.-Y. Liang, C.-K. Shieh, and W. Zhu. Task mapping on distributed shared memory systems using hopfield neural network. In *Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 37–43, 1997.

[50] M. Lin and L. T. Yang. Hybrid genetic algorithms for scheduling partially ordered tasks in a multi-processor environment. In *Proceedings of Sixth International Conference on Real-Time Computing Systems and Applications*, page 382, 1999.

[51] X. Martorell. Blue Gene/L performance tools. *IBM Journal of Research and Development*, 49(2/3):407–424, 2005.

[52] C. W. McCurdy. Creating science-driven computer architecture, 2002. Lawrence Berkeley National Laboratory, `http://www.nersc.gov/news/reports/ArchDevProposal.5.01.pdf`.

[53] P. Merkey. Beowulf history, 2004. `http://www.beowulf.org/overview/history.html`.

[54] M. Miki, T. Hiroyasu, T. Yoshida, and T. Fushimi. Parallel simulated annealing with adaptive temperature determined by genetic algorithm. In *Proceedings of the 2002 IEEE International Conference on Systems, Man, and Cybernetics*, 2002.

[55] S. Moh, C. Yu, H. Y. Youn, B. Lee, and D. Han. Mapping strategies for switch-based cluster systems of irregular topology. In *Proceedings of ICPADS*, pages 733–740, 2001.

[56] D. M. Nicol and W. Mao. On bottleneck partitioning of k-ary n-cubes. *Parallel Processing Letters*, 6:389–399, 1996.

[57] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[58] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, MA, 2nd edition, 1992.

[59] S. Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM News (Newsjournal of the Society for Industrial and Applied Mathematics)*, 38(9):1,3, 2005.

[60] S. Salcedo-Sanz, Y. Xu, and X. Yao. Hybrid meta-heuristics algorithms for task assignment in heterogeneous computing systems. *Computers & Operations Research*, 33(2006):820–835, 2004.

[61] K. B. Sheets, P. S. Smith, S. J. Engel, Y. Deng, J. Guistozzi, and A. Korobka. Method and system for providing dynamic hosted service management across disparate accounts/sites, November 2000.

[62] B. Smith and B. Bode. Performance effects of node mappings on the IBM BlueGene/L machine. In *Proceedings of Euro-Par 2005*, pages 1005–1013, 2005.

[63] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3:85–93, 1977.

[64] N. Suri and A. Mendelson. Design of a parallel interconnect based on communication pattern considerations. *Journal of Parallel Algorithms and Architectures*, 16(4):243–271, 2001.

[65] E-G. Talbi and T. Muntean. A new approach for the mapping problem: A parallel genetic algorithm. In *Proceedings of 2nd Symposium on High Performance Computing*, pages 71–82, 1991.

[66] E.-G. Talbi and T. Muntean. General heuristics for the mapping problem. In *Proceedings of World Transputer Congress*, pages 1229–1241, 1993.

[67] K. Taura and A. Chien. A heuristic algorithm for mapping communicating tasks on heterogeneous resources. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, pages 102–115, Washington, DC, 2000. IEEE Computer Society.

[68] R. A. van de Geign and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[69] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98*, 1998.

[70] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.