Topology-aware task mapping for reducing communication contention on large parallel machines

Tarun Agarwal, Amit Sharma, Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{tagarwal, asharma6, kale}@cs.uiuc.edu

Abstract

Communication latencies constitute a significant factor in the performance of parallel applications. With techniques such as wormhole routing, the variation in no-load latencies became insignificant, i.e., the no-load latencies for far-away processors were not significantly higher (and too small to matter) than those for nearby processors. Contention in the network is then left as the major factor affecting latencies. With networks such as Fat-Trees of hypercubes, with number of wires growing as $P \log P$, even this is not a very significant factor. However, for torus and grid networks now being used in large machines such as BlueGene/L and the Cray XT3, such contention becomes an issue. We quantify the effect of this contention with benchmarks that vary the number of hops traveled by each communicated byte. We then demonstrate a process mapping strategy that minimizes the impact of topology by heuristically minimizing the total number of hop-bytes communicated. This strategy, and its variants, are implemented in an adaptive runtime system in Charm++ and Adaptive MPI, so it is available to many applications written using Charm++ as well as MPI.

1 Introduction

An increasingly large number of scientific pursuits use computational resources as their backbone. Applications range from study of molecular behavior, both using classical and quantum physics models, evaluation of physical properties of materials like stress response, to simulations of galaxies and cosmological phenomenon. The insatiable computational requirements of such applications has inspired the development of massively parallel machines, like the recent BlueGene (BG/L) machine from IBM. Parallelism at the scale of tens of thousands of processors is being seen. For example, BG/L will have 64K processors [1] once fully deployed. The main resources in a large parallel machine are its compute nodes and the interconnection network. It is imperative that techniques for efficient and uniform utilization of these resources be developed.

A parallel program can be thought of as a collection of communicating tasks. Each task has certain computation and communication characteristics. The task assignment problem aims at balancing computational load among the processors in the system and reducing the overhead of communication between them. This requires *partitioning* of tasks into p groups to achieve computational load balance and appropriate *mapping* of these groups onto processors in the network topology to minimize the overhead of comunication. In this paper, we present a heuristic algorithm for solving the mapping problem.

Communication is an important factor in determining performance of parallel programs. Due to the increasing size of the parallel computers being used, the interconnection network has become the system bottleneck. It is so because the packaging considerations for a large number of processors lead to the choice of a mesh or a torus topology. For example, the primary network in BlueGene/L is a 3D-Torus which can be converted to 3D-mesh, if required. Even for a relatively moderate machine size messages might travel a large number of hops on average. For example, a $(16, 16, 16)$3D-Torus on $4k$ processors has a diameter of 24 hops and the average internode distance of 12 hops. If packets travel over such large number of hops, the average load on the links increases, which increases contention. Table 1 presents a simple illustration of this effect. We run a $3D$ Jacobi-like program where elements are logically arranged in a 3D-mesh and send messages to all its neighbours in each iterations. There are 512 elements that are to be mapped onto 512 BlueGene processors connected in a 3D-mesh. We compare the total time taken to complete 200 iterations under the optimal mapping (a simple isomorphism mapping) with that taken under a random mapping for different message sizes. Under the optimal mapping, messages travel only one hop and average load per link is minimized. The reduction in contention leads to faster execution time, with larger gains as message sizes increase. Therefore, it is desirable to map communicating objects to nearby processors.

The task mapping problem is known to be NP-Complete [5, 16, 19]. Two kinds of algorithms have been developed in the past to solve it: Heuristic algorithms [16, 3, 21] and Physical optimization algorithms [2, 6, 18]. Though physical optimization algorithms produce high-quality solutions (better than heuristic algorithms), they

Message Size	Random Mapping	Optimal Mapping
1KB	56.93ms	46.91ms
10KB	243.64ms	124.56ms
100KB	2247.75ms	914.72ms
500KB	11.62s	4.44s
1MB	23.50s	8.80s

Table 1: Time for 200 iterations of a Jacobi-like program with optimal mapping and random mapping

tend to be very slow. Their execution times are unacceptable in a practical scenario for large data sets, when compared to the task execution times. Heuristic algorithms, on the other hand, are much faster and suitable for real-world parallel applications.

To perform topology-aware task mapping, we need to carry out four steps. First, we need to know the communication and computation characteristics of the task in the parallel program. Second, we have to characterize the available system resources (parallel architecture). Third, an evaluation function (or metric) has to be developed to evaluate the solutions. Finally, the mapping technique or heuristic has to be designed to minimize that metric.

The first and second steps are taken care of by the CHARM++ [12] virtualization model and the dynamic load balancing framework [22] implemented in it. The CHARM++ programming model involves breaking up the application into a large number of communicating objects which can be freely mapped to the physical processors by the runtime system [11]. Furthermore, these objects are migratable, which allows the runtime system to perform dynamic load balancing based on measurement of load and communication characteristics during actual execution. This flexibility has been utilized in the dynamic load balancing framework of CHARM++. Dynamic load balancing has an associated overhead of task migration. In CHARM++ this is handled using the PUP framework [10] which is a way of describing the layout of object's data in memory. The metric and the mapping heuristic, which form third and fourth steps, have been described in detail in later sections.

Also note that in this paper, we are only concerned with process-based model [5, 20] in which there are no DAG-based dependencies. The tasks (or modules) are arranged in undirected graphs and edges represent total communication between the tasks at the end points rather than precedence or one-way communication. Further, the tasks are persistent processes which have stable communication patterns between them (CHARM++ Load Balancing Model).

2 Related Work

The problem of scheduling tasks on processors has been well studied. There have been many distinct categories of research, each with a different focus. A large part of the work has concentrated on balancing compute load across the processors while ignoring any communication all together. The problem handled in this kind of work is the assignment of a set of n jobs (each with some arbitrary size) on p processors (n usually larger than than p), so as to minimize the maximum load (makespan) on the processors, since higher compute load on one processor slows down the entire system. In the next category, researchers have worked on communication-sensitive clustering while still ignoring any topology considerations. The main objective here is the partitioning of jobs into balanced groups (equal in number to the number of processors) while reducing inter-partition communication. The more general problem is one of mapping task graph to a network topology graph while balancing compute load on processors and minimizing communication cost (which we model as hop-bytes in section 3). All the categories described involve NP-hard optimization problems. To solve these problems, researchers have made use of heuristic algorithms like greedy, branch-and-bound, local search etc. and physical optimization algorithms like simulated annealing, genetic techniques, neural networks etc. This section will present a brief survey of related works in the third category (mapping task graph on a network topology graph). While keeping communicating tasks on the same processor helps reduce the communication cost, processor computation load considerations prevent all communication from being intra-processor. The problem of mapping communication tasks onto a processor *topology* has been studied in the past. The objective of the mapping is to essentially reduce communication cost by placing communicating tasks on nearby processors.

Bokhari [5] uses the number of edges of the task graph whose end points map to neighbors in the processor graph as the cost metric. The algorithm [5] starts with an initial mapping and performs pairwise exchanges to improve the metric. Results are given for up to 49 tasks. Lee and Aggarwal [16] propose a step by step greedy algorithm followed by an improvement phase. At the first step, the most communicating task is placed on a processor with similar degree. Subsequent placements are guided by an objective function. Berman and Snyder [4] present an approach where both cardinality variation (difference in number of tasks and processors) and topological variations (different in shapes of the task graph and topology graph) are considered. They first coalesce the task graph to get rid of the cardinality variation. The coalesced graph is mapped on the actual topology.

Local search techniques such as Simulated annealing have also been tried. Bollinger and Midkiff [6] propose a two-phased annealing approach: *process annealing* assigns task to processors and *connection annealing* schedules traffic along network links to reduce conflicts. Evolution-inspired Genetic algorithms based search has also been attempted. Arunkumar and Chockalingam [2] propose a genetic approach where search is performed using operators such as *selection*, *mutation*, and *crossover*. While these approaches produce good results, the time required for them to converge is usually quite large compared to the execution time of the application. Orduña, Silla and Duato [18] also propose a variant of the genetic approach. Their scheme starts with a random initial assignment, the *seed*, and in each iteration an exchange is attempted and the gain, if any, is recorded. If no improvement is seen for some iterations a new seed is tried and eventually the best overall mapping is returned.

Strategies for specific topologies and/or specific task graphs have also been studied. Ercal, Ramanujam and Sadayappan [7] provide a solution in the context of hypercube topology. Their divide-and-conquer technique, called *Allocation by Recursive Mincut* or ARM, aims to minimize total inter-processor communication subject to the processor load being within a tolerance away from the average. A mincut is calculated on the task graph while maintaining processor load equal on the two sides and a partial assignment of the two parts is made. Repetitive recursive bipartitioning is performed and the partition at the k^{th} iteration determines the k^{th} bit of the processor assignment. Bianchini and Shen [9] consider mesh network topology. Fang, Li and Ni [8] study the problem of 2-D convolution on mesh, hypercube and shuffle-exchange topologies only.

Baba, Iwamoto and Yoshinaga [3] present a group of mapping heuristics for greedy mapping of tasks to processors. At each iteration a task is selected based on a heuristic, and then a processor is selected for that task based on another heuristic. One of the more promising heuristic combinations they propose is to select the task that has maximum total communication with already assigned tasks and place it on the processor where the communication cost is minimized. The communication cost is modeled similar to hop-bytes, although considering only the communication with previously assigned tasks. A very similar scheme has also been implemented, independently, in CHARM++ as the TopoCentLB load balancing strategy. Taura and Chien [21] propose a mapping scheme in the context of heterogeneous systems with variable processor and link capacities. In their scheme tasks are linearly ordered with more communicating tasks placed closer, and the tasks are mapped in this order.

3 Definitions

Both the load information and the network topology are represented as graphs.

- **Topology Graph** The network topology is represented as an undirected graph $G_p = (V_p, E_p)$ on p $(= |V_p|)$ vertices. Each vertex in V_p represents a processor, and an edge in E_p represents a direct link in the network. Our algorithms work for arbitrary network topologies; however we will present results on more popular topologies like Torus and Mesh.

- **Task Graph** The parallel application is represented as a weighted undirected graph $G_t = (V_t, E_t)$. The vertices in V_t represent compute objects (or groups of objects) and the edges in E_t represent direct communication between the compute objects (or groups of objects). Each vertex $v_t \in V_t$ has a weight $\hat{w_t}$. The weight on a vertex denotes the amount of *computation* that the objects in the vertex represent. Similarly, each edge $e_{ab} = (v_a, v_b) \in E_t$ has a weight c_{ab}. The weight c_{ab} represents the amount of *communication* in bytes between the compute objects represented by v_a and v_b.

- **Task Mapping** The task-mapping is represented by a map :
$$P : V_t \longrightarrow V_p$$
If the compute objects represented by the vertex $v_t \in V_t$ of the task-graph are placed on processor v_p, then $P(v_t) = v_p$. A **partial task mapping** is one where some of the vertices of the task-graph have been assigned to processors in the topology-graph while others are yet to be assigned. A partial mapping can be represented by a function :
$$P : V_t \longrightarrow V_p \cup \{\perp\}$$
where $P(v_t) = \perp$ denotes that v_t has not yet been assigned to a physical processor.

- **Hop-bytes (Metric)** Hop-bytes is the metric (or evaluation function) used to judge the quality of the solution produced by the mapping algorithm. Hop-bytes is the total size of inter-processor communication in bytes weighted by distance between the respective end-processors. The relevant measure for distance between two processors is the length of the shortest path between them in the topology-graph. For processors $p_1, p_2 \in V_p$, the distance between them is represented by $d_p(p_1, p_2)$. Let us denote by $HB(G_t, G_p, P)$ the hop-bytes when the task graph G_t is mapped on the topology graph G_p, under the mapping P.
$$HB(G_t, G_p, P) = \sum_{e_{ab} \in E_t} HB(e_{ab})$$
$$where\ HB(e_{ab}) = c_{ab} \times d_p(P(v_a), P(v_b))$$
The overall Hop-bytes is the sum of Hop-bytes due to individual nodes in the task graph.
$$HB(G_t, G_p, P) = \frac{1}{2} \sum_{v_a \in V_t} HB(v_a)$$
$$where\ HB(v_a) = \sum_{e_{ab} \in E_t} HB(e_{ab})$$

- **Hops per byte** This is the average number of network links a byte has to travel under a task mapping.
$$Hops\ per\ Byte = \frac{HB(G_t, G_p, P)}{\sum_{e_{ab} \in E_t} c_{ab}}$$
$$Hops\ per\ Byte = \frac{\sum_{e_{ab} \in E_t} c_{ab} \times d_p(P(v_a), P(v_b))}{\sum_{e_{ab} \in E_t} c_{ab}}$$

4 The mapping heuristic

Assume we have n compute objects and p processors. The problem of balancing compute load involves partitioning the n compute objects into p groups such that the total compute load of objects in each group is roughly the same. The second problem, that of reducing network contention, involves placing these groups onto the p processors such that more heavily communicating groups are placed on nearby processors. This would make each message travel over a smaller number of links leading to a reduction in the average data transferred across individual links.

The problems of partitioning and mapping can either be solved together or in separate phases. In the latter approach, the first phase, called the *partitioning phase*, involves partitioning the objects (oblivious to network-topology) into p groups. This serves the objective of balancing compute load on processors. In the next phase, the *mapping phase*, the p groups are mapped onto the p processors with the objective of placing communicating groups on nearby processors. Any partitioning algorithm can be used in the *partitioning phase*. However, a partitioning method that reduces inter-group communication by placing heavily communicating objects in the same group must be preferred. This two-phased approach has the advantage of simplicity and clear separation of the two objectives. A unified approach where the mapping is performed on an object-by-object basis has more freedom but suffers from the constraint of balancing the compute load on processors. The additional constraint makes this approach more complex. We have adopted the above mentioned two-phased approach in this paper.

We now present the mapping heuristic. It is applied in the second phase of the two-phased approach. The partitioning in the first phase is accomplished either using METIS [13, 15, 14] or using some of the existing topology-oblivious load balancing strategies in CHARM++.

4.1 Intuition

We employ an iterative approach in mapping tasks to processors. In this approach, the main question that needs to be addressed is the selection of the next processor and the next node in the task-graph to be placed on it. This is guided by an *Estimation function*. It estimates for each pair of unallocated tasks and available processors the *cost* of placing the task on the processor in the next cycle. The estimation function has the following form:

$$f_{est}(t, p, P) \longrightarrow cost\ value$$

where t is an unassigned task , p is an available processor and P is the current task mapping. For each task we can find the best processor, the one where it costs least to place it. However, for a given task it may not matter much if it is placed on its best processor or any other processor. We can approximate how *critical* it is to place a task by assuming that if it is not placed in the next cycle it will go to some arbitrary processor in a future cycle. The estimation function gives us the cost of placing a task on its best processor and the expected cost when placed on an arbitrary processor. The difference in the two values is used as a measure of how critical it is to place the task in the next cycle. Once we estimate how critical it is for each task to be placed in the next cycle, we can select the one for which it is most critical.

Algorithm 1: The Mapping Algorithm

```
begin
    Data: V_t (the set of Tasks),
          V_p (the set of processors)
          (|V_t| = |V_p| = n)
    Result: P : V_t ⟶ V_p (A task mapping)

    T_1 ⟵ V_t;
    P_1 ⟵ V_p;
    for k ← 1 to n do
        //Select the next task and processor (t_k, p_k);
        //Next task, t_k, is the one with maximum
        gain;
        max_gain ← −∞;
        for task t ∈ T_k do
            gain(t) =
            (Σ_{p∈P_k} f_est(t,p)) / (n−k) − min_{p∈P_k} f_est(t, p);
            if gain(t) > max_gain then
                t_k ← t;
                max_gain ← gain(t);
        end
        //Next processor, p_k, is the one where t_k
        costs least;
        min_cost ← ∞;
        for processor p ∈ P_k do
            if f_est(t_k, p) < min_cost then
                p_k ← p;
                min_cost ← f_est(t_k, p)
        end

        P(t_k) = p_k;

        T_{k+1} ← T_k − {t_k};
        P_{k+1} ← P_k − {p_k};
end
```

4.2 The algorithm

The top-level view of the algorithm is shown as algorithm 1.

Let us denote by T_k the set of tasks that remain to be placed at the beginning of the k^{th} cycle. Also denote by P_k the set of processors that are available at the beginning of the k^{th} cycle. As shown in Algorithm 1, we calculate the estimated gain which each task stands to achieve if it is placed in the current cycle. The estimation function is such that $f_{est}(t, p, P)$ approximates the contribution of task t (if placed on processor p) to overall quality of the mapping. The function is topology-sensitive. Once gain values are known for each task, the one with maximum gain is selected. It is mapped to the processor where f_{est} estimates it to cost the least.

4.3 Estimation functions

In this section we will motivate and present multiple cost estimation functions. As explained earlier the estimation function is used for calculating the *cost* of placing a task t on an available processor p when some of the tasks have already been placed. Since our objective is to reduce hop-bytes, we would interpret the contribution of task t to overall Hop-bytes as the *cost* of placing t on processor p. Let us recall that $G_t = (V_t, E_t)$ is the task graph and $G_p = (V_p, E_p)$ is the network topology graph. We note that the overall Hop-bytes is additive and is the sum of the Hop-bytes due to individual tasks.

$$HB = \sum_{e_{ij}=(t_i,t_j)\in E_t} c_{ij} d_p(P(t_i), P(t_j)) = \frac{1}{2} \sum_{t_i \in V_t} HB(t_i),$$

$$where \quad HB(t_i) = \sum_{t_j|(t_i,t_j)\in E_t} c_{ij} d_p(P(t_i), P(t_j))$$

During a particular iteration of the mapping algorithm, we only have a partial mapping because some tasks have not been placed yet. Let T_k be the set of tasks that remain to be placed and P_k be the set of processors that are available at the beginning of the k^{th} iteration. Similarly, let $\bar{T_k}$ be the set of tasks that have already been placed and $\bar{P_k}$ be the set of processors that are no longer available at the k^{th} iteration. Note that $T_k \cap \bar{T_k} = \phi$ and $P_k \cap \bar{P_k} = \phi$. Also, they partition the complete sets, which can be stated as : $T_k \cup \bar{T_k} = V_t$ and $P_k \cup \bar{P_k} = V_p$.

1. *First order approximation*
   Since we do not know the placement of some of the tasks yet, we drop terms corresponding to those tasks. Thus, we consider the contribution only due to communication with already assigned tasks:

   $$f_{est}(t_i, p, P) = \sum_{t_j \in \bar{T_k}} c_{ij} d_p(p, P(t_j))$$

   It is quite cheap to compute as compared to the other approximations. This estimation function has been used in TopoCentLB described in 4.5.

2. *Second order approximation*
   We will approximate the contribution of communication with tasks that have not yet been assigned. As we do not yet know the placement of an unassigned task, say t_j, in T_k, we assume that it will be placed on a random processor. Thus, we approximate the distance between p and $P(t_j)$ by the *expected* distance of p to other processors. The distribution of $P(t_j)$ is taken to be uniformly random on P_k. In other words, for any unmapped task $t_j \in T_k$ we approximate:

   $$d_p(p, P(t_j)) \approx E_{p_j \in U[V_p]}[d_p(p, p_j)] = \frac{\sum_{p_j \in V_p} d_p(p, p_j)}{|V_p|}$$

   Thus we can refine our estimation function to be:

   $$f_{est}(t_i, p, P) = \sum_{t_j \in \bar{T_k}} c_{ij} d_p(p, P(t_j))$$

   $$+ \sum_{t_j \in T_k} c_{ij} \frac{\sum_{p_j \in V_p} d_p(p, p_j)}{|V_p|}$$

3. *Third order approximation*
   While we do not yet know the placement of unassigned tasks, we do know that they can only be assigned to processors that are still available. The approximation that an unassigned task, say t_j, will be mapped to a random processor in V_p does not capture this constraint. We should rather assume the distribution of $P[t_j]$ to be uniformly random on *available* processors P_k. In other words, for any unmapped task $t_j \in T_k$ we approximate:

   $$d_p(p, P(t_j)) \approx E_{p_j \in U[P_k]}[d_p(p, p_j)] = \frac{\sum_{p_j \in P_k} d_p(p, p_j)}{|P_k|}$$

While using a better approximation in the estimation function (in the third order approximation) is expected to lead to a better solution, it is costlier to compute and it affects the overall running time of the load balancing algorithm (see section 4.4). Since the consideration of running time dominates in the real-world applications, we will use the second order approximation scheme in our implementation and results. This will be discussed in section 4.4.

4.4 Implementation of the algorithm: TopoLB

The mapping algorithm has been implemented in CHARM++ as a strategy called TopoLB under the dynamic load-balancing framework. Initially, the task graph is partitioned into p groups using METIS. Any other topology-oblivious partitioner can also be specified for partitioning. Some of the dynamic load balancing strategies of CHARM++ like GreedyLB are suitable for partitioning. At this point, both the new task graph and the topology graph have the same size p. During the iterations of the algorithm, we maintain a $p \times p$ table of dynamic values of $f_{est}(t, p, P)$. Rows are indexed by task nodes and columns are indexed by processors. The entry in the cell (t, p) is the current value of $f_{est}(t, p, P)$. In addition, we maintain the minimum and average value of f_{est} for each unassigned task over all unassigned processors. Let us call these arrays $FMin[t]$ and $FAvg[t]$, respectively. In the k^{th} iteration we need to select the unassigned task t_k, which maximizes the value of $FAvg[t] - FMin[t]$. This takes a linear pass, taking time $O(p)$. Next we find the available processor p_k, where $f_{est}(t_k, p, P)$ attains the minimum value in time $O(p)$. The task t_k is mapped to processor p_k which is marked unavailable. The main cost is incurred in updating the table at the end of each iteration, as f_{est} values might change as a result of the assignment of t_k to p_k. Here, we discuss the time-complexity only for the second and third order approximations. In the second order approximation, only the estimation values of tasks that have an edge with t_k in the task graph are affected. Moreover, updating the f_{est} values for one such task takes a total of $O(p)$. This makes the total cost of update $O(p\delta(t_k))$, where $\delta(t_k)$ denotes the degree of the node t_k in the task graph. Thus, the total time in each iteration of the algorithm is $O(p)+O(p\delta(t_k))$, which is same as $O(p\delta(t_k))$. The total running time over all p iterations is:

$$Running\,Time = \sum_{t \in V_t} O(p\delta(t)) = O(p \sum_{t \in V_t} \delta(t)) = O(p|E_t|)$$

While the running time $O(p|E_t|)$ can be as high as $O(p^3)$, in practice the nodes in the task graph have small constant degree, and a running time closer to $O(p^2)$ is observed. In the third order approximation, however, the value $f_{est}(t, p)$ depends on the average distance of processor p to other *free* processors. When the status of p_k changes from free to allocated, the average changes for all other processors. Thus, all $f_{est}(t, p, P)$ values change. By maintaining the average distance of a processor to free processors, we incur a constant cost per processor in calculating new average values; this is a total cost of $O(p)$. Once average distances are known, each value in the f_{est} table can be updated in constant time. This incurs a total cost of $O(p^2)$. Thus total time in an iteration is $O(p) + O(p^2)$, which is same as $O(p^2)$. Overall running time over all p iterations in this case is:

$$Running\ Time = \sum_{t \in V_t} O(p^2) = O(p^3)$$

From the above calculation we can see that using second order approximation ($O(p|E_t|)$) takes less time than third order approximation ($O(p^3)$). In practice, the nodes of the task graph have a small constant degree, and the total number of edges is $O(p)$. Thus, the second order approximation has a running time closer to $O(p^2)$ which is significantly lower than the fixed cost of $O(p^3)$ for the third order approximation. Scaling considerations lead us to the choice of second order approximation for our scheme.

4.5 TopoCentLB

TopoCentLB is a topology-aware load balancing strategy for CHARM++ which also tries to solve the task mapping problem. In this strategy, as in TopoLB, the original task graph is first partitioned using a topology-oblivious scheme (like greedy partitioning or Metis) to get a smaller graph with p nodes, where p is the number of physical processors. We will assume for the description that the task graph and the processor graph have the same sizes. The mapping algorithm iteratively maps the nodes of this task graph onto the physical processor graph. In the first iteration, the most communicating task is selected and mapped to a processor. In each subsequent iteration, the task that has maximum total communication with already assigned tasks is selected. It is mapped to the free physical processor where it incurs the least total cost of communication (in terms of hop-bytes) with the already assigned tasks. Thus, the algorithm uses first order approximation to the estimation function described earlier. However, its choice in each iteration depends on the estimated cost itself, while TopoLB selects the task whose placement is most critical. A similar strategy has been described by T. Baba et.al. [3]; where it corresponds to their (P_3, P_4) scheme.

We discuss the time complexity of TopoCentLB. The algorithm is implemented using heap data structure. In the k^{th} iteration, the selection of task t_k involves extraction of t_k from the heap and updation of keys of the neighbors of t_k which are in the heap. Extraction and updation both take $log(p)$ time. Hence,it is bounded by $O(log(p) + log(p)\delta(t_k))$ where $\delta(t_k)$ is the degree of t_k in the task graph. To place t_k on a processor, we go over all the unassigned processors. For each unassigned processor p_k, we calculate the amount of communication of t_k with its assigned neighbors to finally arrive at the minimum value of communication. Hence the cost involved is bounded by $O(p\delta(t_k))$. So, the total running time of the algorithm is:

$$Running\ Time = \sum_{t \in V_t} O((log(p) + p)\delta(t))$$

$$= O(p \sum_{t \in V_t} \delta(t)) = O(p|E_t|)$$

5 Experiments

In this section we will discuss and compare the performance of the load balancing schemes described earlier. We also compare their performances to a load balancer which places the tasks on the processors at random. Section 5.2 will describe the performance of TopoLB in reducing the hops-per-byte metric in different scenarios. The effect of the reduction in hops-per-byte on actual network communication observables, like average message latency and execution times , is described in section 5.3.

5.1 Evaluation mechanism

CHARM++ load balancing framework allows the runtime to log load information from an actual parallel execution into a file for later analysis. This can be done by specifying the load balancing step for which the load information needs to be logged as runtime parameters (using +LBDump *StartStep* to specify the first step, and +LBDumpSteps *NumSteps* to specify the total number of steps). A log file is generated for each of the steps specified in the range. The effect of different centralized load balancing strategies can then be studied on the load balancing database present in these log files by running any CHARM++ program sequentially in simulation mode (by specifying the name using +LBDumpFile *FileName* and the load balancing step to be simulated using +LBSim *StepNum*). In simulation mode, the load balancing framework uses the load information from the log files rather than from the current run. Relevant metrics can be studied as needed.

This mechanism provides an efficient way of testing load balancing strategies as their effects on a given load scenario can be studied without repeated runs of the actual parallel program. Moreover, different strategies can be compared on exactly the same load scenarios, which is not possible in actual execution because of non-deterministic interleaving of events. Thus, we will use this mechanism to study the performance of the load balancing schemes described earlier.

5.2 Reduction in hop-bytes

As described in section 4, the metric that the mapping heuristic (TopoLB) aims to reduce is hop-bytes, or equivalently, hops-per-byte. We will present the performance in terms of hop-bytes reduction.

To study the quality of mapping independent of the partitioning method, we can start with task graphs that have just p tasks so that no clustering is needed. We use a CHARM++ benchmark program which has a jacobi-like communication pattern for this purpose. The benchmark program creates chares (or tasks) which communicate in a 2D-Mesh pattern. Each chare communicates with its four neighbors (three or two for boundary and corner chares, respectively) in each iteration. The number of chares to be created is a parameter to the benchmark.

The number of processors involved in our study is quite high. We emulate this large number of processors using the Bluegene version of CHARM++. CHARM++ can be built such that all the CHARM++ application programs run on top of a Bluegene emulator. This emulator allows us to emulate a large number of Bluegene processors. Our emulator gives us the flexibility of connecting the Bluegene processors in many different topologies.

5.2.1 2D-Mesh pattern on 2D-Torus

Figure 1 compares the performance of random placement, TopoLB and TopoCentLB in mapping a 2D-Mesh pattern onto a 2D-Torus topology. In each case, the number of tasks created is the same as the number of processors. It can be seen that random placement produces mappings that have very large values of hops-per-byte. We can analytically compute the expected hops-per-byte for random placement, which is same as the expected distance between two random processors. Each dimension has a span of $\sqrt{p}$, and with a wrap-around link the expected distance in each dimension is $\frac{\sqrt{p}}{4}$. Thus, the total expected distance between two random processors is $2\frac{\sqrt{p}}{4}$, or $\frac{\sqrt{p}}{2}$. As seen in Figure 1, the value of hop-bytes for random placement matches closely with this expected value.

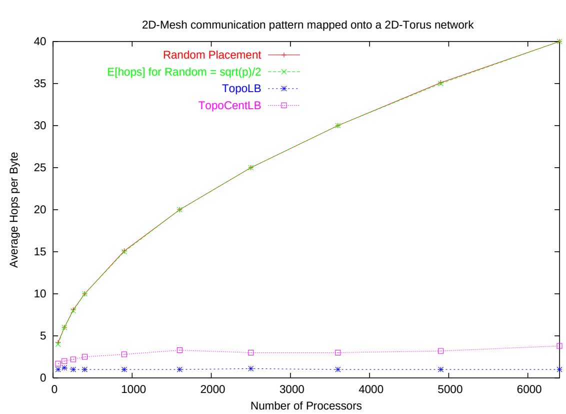

Figure 1: Mapping 2D-Mesh communication pattern onto a 2d-Torus. Random placement matches expected value.

Since a 2D-Torus contains a 2D-Mesh, the ideal placement can preserve neighborhood relationships and achieve the hops-per-byte value of 1. It is interesting to note that TopoLB actually produces an optimal mapping in most cases. Figure 2 shows the comparison of TopoLB and TopoCentLB and is essentially a zoomed-in version of figure 1. It is also seen that TopoCentLB also results in small values of hops-per-byte, though TopoLB performs better than TopoCentLB in all tested cases.

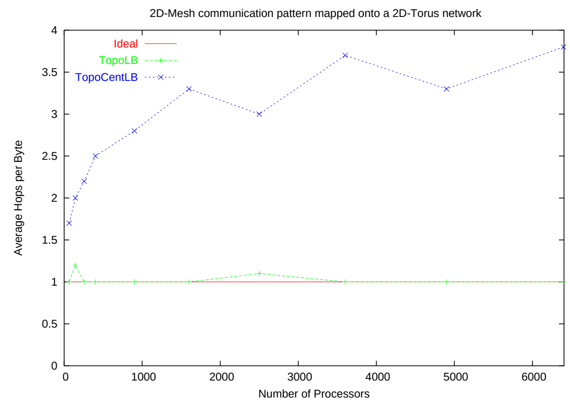

Figure 2: Mapping 2D-Mesh communication pattern onto a 2d-Torus. Zoomed in to compare TopoLB and TopoCentLB.

5.2.2 2D-Mesh pattern on 3D-Torus

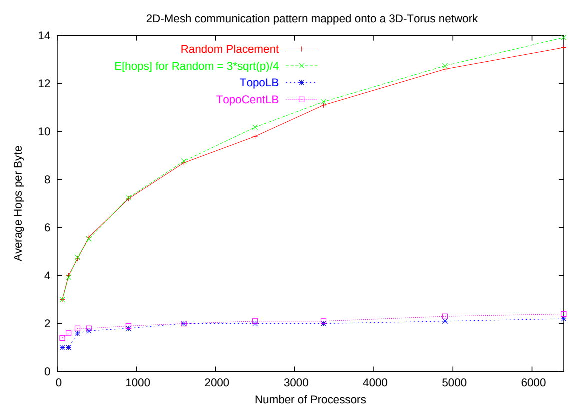

Figure 3: Mapping 2D-Mesh communication pattern onto a 3d-Torus. Random placement matches expected value.

Next we map the 2D-mesh communication pattern on a 3D-Torus topology of the same size. A comparison of the average hops-per-byte values resulting from different mapping strategies is shown in figure 3. For a 3D-Torus, the expected distance between two random processors is $3\frac{\sqrt[3]{p}}{4}$. As seen in figure 3, the actual value of hops-per-byte obtained by random mapping matches this analytical formula closely. The other two mapping strategies, TopoLB and TopoCentLB, lead to considerable reduction in hops-per-byte when compared to a random mapping.

In general, the task graph (2D-Mesh) is not a subgraph of the topology graph (3D-Torus). Hence, it is not always even feasible to preserve neighborhood relation when map-

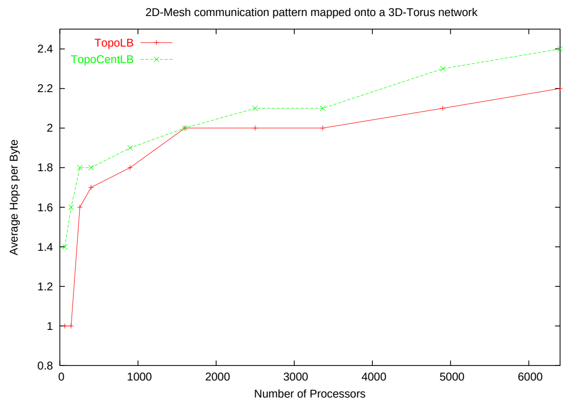

Figure 4: Mapping 2D-Mesh communication pattern onto a 3d-Torus. Zoomed in to compare TopoLB and TopoCentLB.

ping a 2D-Mesh onto a 3D-Torus with the same number of nodes. Consequently, the optimal value of hops-per-byte is, in general, larger than 1. However, for specific cases, it is possible to preserve the neighborhood relation. For example, a (8,8)2D-Mesh is a subgraph of a (4,4,4)3D-Torus, so it is possible to preserve neighborhood relation. We can see from figure 4 that in this case, TopoLB is able to reduce hops-per-byte to its optimal value of 1 (the value when number of processors is 64 in the figure). For a larger number of processors, TopoLB leads to a small value of hops-per-byte. TopoCentLB also results in small values of hops-per-byte which are about 10% higher than those from TopoLB.

5.2.3 LeanMD mapped onto different topologies

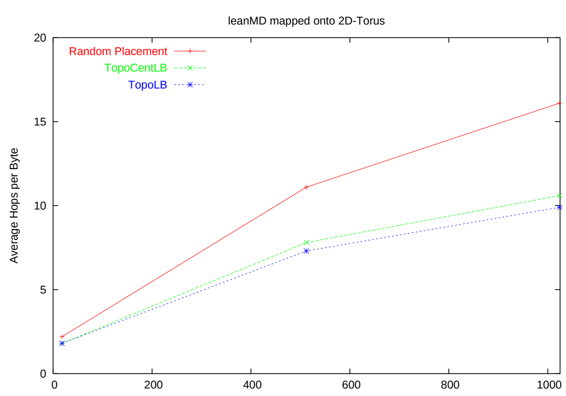

Figure 5: Comparison of different mapping strategies on 2D-tori for LeanMD data

This section will describe the results of mapping communication pattern from a real molecular dynamics simulation program called LeanMD [17]. We have load information dumps for LeanMD on different numbers of processors. The total number of chares is $3240 + p$ where p is the number of processors. This gives virtualization ratios of 180 for $p = 18$, 6 for $p = 512$ and 3 for $p = 1024$. Since the number of chares is greater than the number of objects, we need to perform clustering of chares into p groups with balanced communication load. We use METIS for this initial grouping. Once this grouping is performed on the original task graph, a new task graph with the same size as the number of processors is obtained. We then map this task graph using different strategies.

Figure 5 shows the average hops-per-byte when LeanMD is mapped onto 2D-Tori of various sizes. For $p = 18$, the virtualization ratio is 180, which is quite high. Consequently, with such a large number of chares in each group, almost all pairs of groups communicate with each other. The average degree of the coalesced task-graph obtained from METIS is 12.7, which means that each group communicates with 70% of the groups. Hence it is difficult for any strategy to reduce hop-bytes as almost all the groups communicate. For 512 processors, the virtualization ration is 6 and the average degree of the coalesced task graph is 19.5 which means that each group communicates with about 4% of the other groups. This creates some avenues for intelligent placement of groups to keep the communication local.

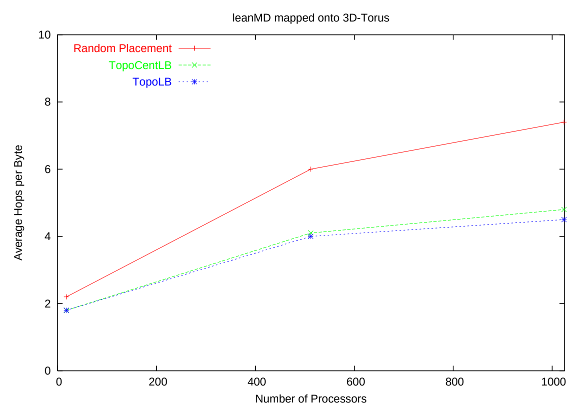

Figure 6: Comparison of different mapping strategies on 3D-tori for LeanMD data

As seen from figure 5, TopoLB leads to a 34% reduction in average hops-per-byte over random placement. A topology-based refiner (implemented in CHARM++ load-balancing framework) called RefineTopoLB can further reduce the value by about 12%. TopoCentLB also performs well, leading to a 30% reduction; similar trend is seen for 1024 processors. Note that RefineTopoLB is intended to be used for further reducing hop-bytes after applying the initial load balancer like TopoLB. The refiner swaps tasks between processors to see if hop-bytes are reduced or not. It swaps only when hop-bytes get reduced.

Figure 6 shows the results for mapping onto 3D-Tori. The relative performance of the different schemes in this case is similar to the last case. TopoLB followed by RefineTopoLB leads to a reduction in hops-per-bytes in the 40% range.

5.3 Network Simulation

In section 5.2 we discussed the reduction in the average number of hops that each byte travels over the network. In this section we will discuss how this reduction in the hops-per-byte metric translates into gains in execution time and

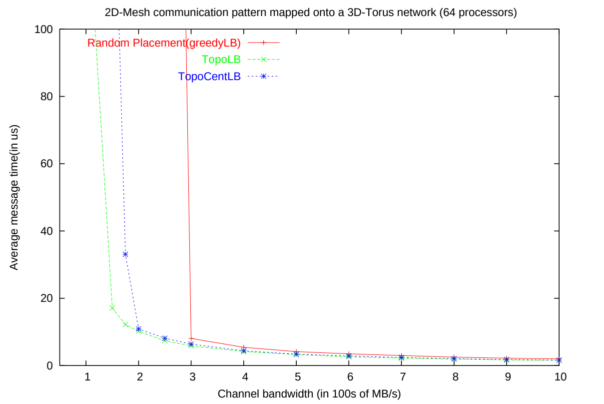

Figure 7: 2D-mesh on 64-node 3D-Torus: Average message latency using different mappings

and other characteristics on the network.

We will perform simulations using BigNetSim [23], which is an interconnection network simulator. One of the features of BigNetSim is that it can simulate application traces on different kinds of interconnection networks. We will be using a 3D-Torus network to simulate a 2D-jacobi like program. In this benchmark program, each chare performs some computation and then sends messages to its four neighbors in each iteration. The amount of computation is kept low so that communication is a significant factor in overall efficiency. This benchmark program is executed with TopoLB, TopoCentLB, and GreedyLB (a CHARM++ load-balancer with essentially random placement) and event traces are obtained. These event traces contain timestamps for message sending and entry point (message receiving) initiation. Event-dependency information is also available in the traces so that these timestamps can be corrected depending on the network being simulated while honoring event ordering. Thus, we can vary the parameters for the underlying interconnection networks and examine the expected effect on the execution of the traced program.

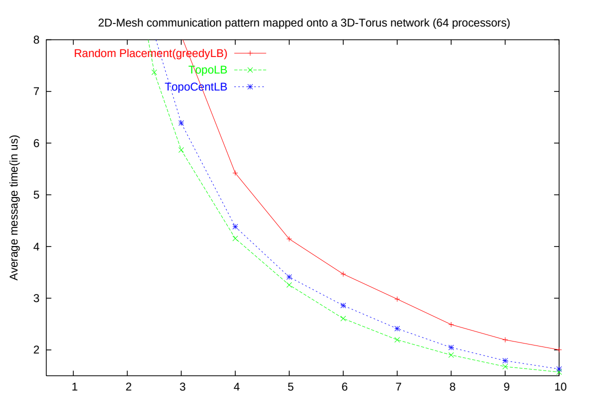

Figure 8: 2D-mesh on 64-node 3D-Torus: Detailed comparative view of average latency in the un-congested domain

The execution of application traces is simulated on a (4,4,4)3D-Torus interconnection network. Since TopoLB and TopoCentLB lead to a reduction in the average hops that a packet travels, the actual network load (and contention) generated for the same application is reduced. Hence, it is expected that an application mapped using these schemes would be able to tolerate reduction in link bandwidth better than a naive random mapping. Figure 7 shows the average message latency for different values of link bandwidth. It can be seen that in the case of a random placement, the average latency increases dramatically as congestion sets in due to a reduction in bandwidth. TopoCentLB can tolerate a further reduction in network bandwidth while TopoLB is the most resilient; this is because a smaller value of hops-per-byte leads to a smaller number of packets on each link. Consequently, the links can service the traffic with a smaller bandwidth. In the case of random placement, larger loads on individual links lead to messages being stranded in the buffers at the switches for a longer time. Figure 8 shows the zoomed-in view of figure 7 for the purpose of comparison of the schemes in the low congestion region. Even in this case, it can be seen that among the three schemes TopoLB leads to least average message latency.

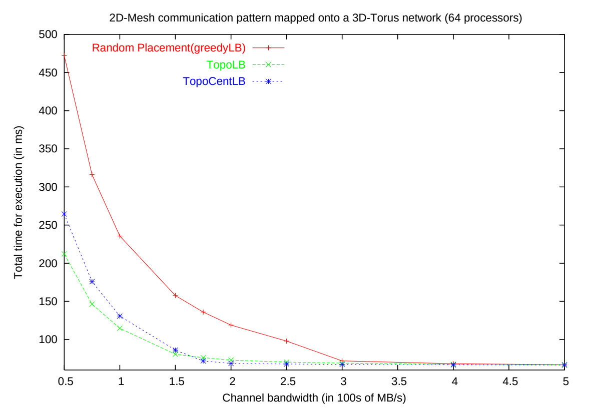

Figure 9: Completion time for the execution of 2000 iterations

The total time for the entire execution to finish is also improved by using intelligent mapping. Figure 9 shows the total time required for the completion of 2000 iterations of the benchmark. For smaller bandwidth, optimizations obtained by TopoLB and TopoCentLB show a very large gain. In this region, random placement leads to congestion which causes communication to be delayed and iterations progress much slower. Total execution time under random placement can be more than double the time required under TopoLB. TopoCentLB also leads to a large reduction over random placement. However, TopoLB outperforms TopoCentLB by about 10-25%.

5.4 Results on Bluegene

In this section, we present performance results on Bluegene [1]. As earlier, we use a 2D Jacobi-like benchmark program. Elements are arranged logically in a 2D Mesh. In each iteration, every element performs some computation and sends a message to each of its four neighbors. The actual network topology in which the physical Blugene processors are connected can be configured as either a 3D-Mesh or a 3D-Torus. We present results on both these network topologies.

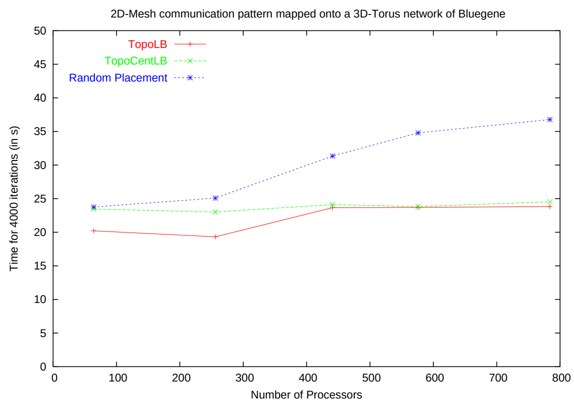

Figure 10: Comparison of mapping strategies on Bluegene 3D-Torus network.

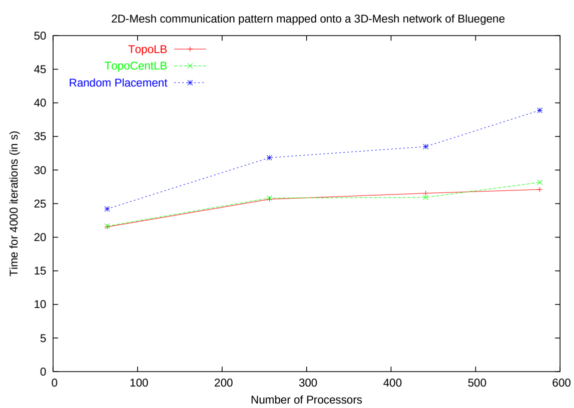

Figure 11: Comparison of mapping strategies on Bluegene 3D-Mesh network.

Figures 10 and 11 compare the time required to complete 4000 iterations of the benchmark for different mapping strategies. The size of messages sent in each iteration is 100KB. This makes the communication to computation ratio high. We can see that both TopoLB and TopoCentLB lead to reduction in time when compared to random mapping. Note that the number of elements is same as the number of processors, so the computational load on processors is balanced. The reduction in execution time can be attributed to communication optimizations.

It can be seen from the figures that the total time required under Mesh connection is generally higher than that for Torus connection. This is because there are additional wrap-around paths in a torus network which help in keeping average load on links lower. However, the effect is more pronounced for random placement than the other two strategies. This may be because random placement leads to long-range messages while TopoLB and TopoCentLB maps elements such that most messages travel over only a small number of hops. If messages travel over a very small number of hops, removal of wrap-around links does not affect the distance travelled by messages in most cases.

6 Conclusions and future work

We presented a heuristic aimed at solving the task mapping problem that arises in the context of parallel computing.

Our heuristic algorithm provides a solution to the problem of mapping tasks onto physical processors connected in a given topology, so that most of the communication occurs between nearby processors. We show that TopoLB provides a good mapping in terms of average number of hops travelled by each byte, and compares favorably with some other schemes. In particular, we found that TopoLB was able to map a 2D-Mesh onto a 2D-Torus almost optimally in many cases, although it does not consider the shapes of the graphs specifically. We also developed another similar, but simpler and faster, scheme called TopoCentLB for the purpose of comparison of its results with TopoLB. We have shown, via simulations, that an efficient mapping which reduces the total communication load on the network, or hop-bytes, leads to lower network latencies on average, and provides better tolerance to network bandwidth constraints and network contention. We validate this conclusion with experiments on Bluegene where we find that communication-intensive programs can be made more efficient with good mappings.

Due to the massively large sizes of machines like Blugene, a distributed approach toward keeping communication localized in a neighborhood may be needed for scalability in the future. Hybrid approaches (semi-distributed), such as that in [22], may also prove effective and need to be investigated further.

References

[1] An Overview of the BlueGene/L Supercomputer. In *Supercomputing 2002 Technical Papers*, Baltimore, Maryland, 2002. The BlueGene/L Team, IBM and Lawrence Livermore National Laboratory.

[2] S. Arunkumar and T. Chockalingam. Randomized heuristics for the mapping problem. *International Journal of High Speed Computing (IJHSC)*, 4(4):289–300, Dec. 1992.

[3] 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, USA, 1990. IEEE Computer Society.

[4] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *J. Parallel Distrib. Comput.*, 4(5):439–458, 1987.

[5] S. H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[6] S. W. Bollinger and S. F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *ICPP (1)*, pages 1–7, 1988.

[7] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 210–221, New York, NY, USA, 1988. ACM Press.

[8] Z. Fang, X. Li, and L. M. Ni. On the communication complexity of generalized 2-d convolution on array processors. *IEEE Trans. Comput.*, 38(2):184–194, 1989.

[9] R. P. B. Jr. and J. P. Shen. Interprocessor traffic scheduling algorithm for multiple-processor networks. *IEEE Trans. Computers*, 36(4):396–409, 1987.

[10] R. Jyothi, O. S. Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[11] L. V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[12] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[14] G. Karypis and V. Kumar. Multilevel algorithms for multiconstraint graph partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[15] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96 – 129, 1998.

[16] S.-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. Computers*, 36(4):433–442, 1987.

[17] V. Mehta. Leanmd: A charm++ framework for high performance molecular dynamics simulation on large parallel machines. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[18] J. M. Orduña, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. In *30th International Workshops on Parallel Processing (ICPP 2001 Workshops), Valencia, Spain*, pages 349–354, 3-7 September 2001.

[19] P. Sadayappan. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Trans. Computers*, 36(12):1408–1424, 1987.

[20] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Engineering*, 3:85–93, Jan. 1977.

[21] 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 (HCW '00)*, page 102, Washington, DC, USA, 2000. IEEE Computer Society.

[22] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 2005.

[23] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, number to appear, 2005.