Topology-aware Task Mapping for Reducing Communication Contention on Large Parallel Machines

Tarun Agarwal, Amit Sharma, Laxmikant V. Kalé
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 for a broad class of applications.

1 Introduction

An increasingly large number of scientific pursuits use computational methods as their backbone. Applications range from study of molecular behavior, evaluation of physical properties of materials, to simulations of galaxies and cosmological phenomenon. The insatiable computational requirements of such applications have inspired the development of massively parallel machines. For example, BlueGene (BG/L) machine from IBM has 64K nodes [1]. 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.

This work was supported in part by the National Science Foundation (ITR 0205611, ITR 0121357) and Department of Energy ASCI center, CSAR (B341494, B505214).

A parallel program can be thought of as a collection of communicating objects. Here, by *object*, we mean a fine-grained task. Each object has certain computation and communication characteristics. The object 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 objects into groups (which we'll call *tasks*) to achieve computational load balance and appropriate *mapping* of these tasks 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. We'll use the above specified notion of *object* and *task* throughout the paper.

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. The packaging considerations for a large number of processors often leads 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 a 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-relaxation (7-point stencil) program where elements are logically arranged in a 3D-mesh and each sends messages to all its neighbours in each iteration. There are 512 elements that are to be mapped onto 512 BG/L 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. To illustrate the communication issues, we set the computation time to zero. Under the optimal mapping, messages travel only one hop and average load per link is minimized. The reduc-

1-4244-0054-6/06/$20.00

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

tion 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.

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 objects 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++ [11, 12] virtualization model, the runtime instrumentation and the dynamic load balancing frameworks [19] implemented in it. The metric and the mapping heuristic have been described in detail in later sections.

In this paper, we are only concerned with the process-based model [5, 17] in which there are no dependencies. The objects are arranged in undirected graphs and edges represent total communication between the objects at the end points rather than precedence or one-way communication. Further, the objects are persistent processes which have stable communication patterns between them.

2 Related Work

The problem of scheduling tasks on processors has been well studied. A large part of the work has concentrated on balancing compute load across the processors while ignoring any communication all together. The problem here is the assignment of a set of n jobs or objects (each with some arbitrary load) on p processors, so as to minimize the maximum load (makespan) on the processors. In the next category, researchers have worked on communication-sensitive clustering while still ignoring any topology considerations [13]. 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 a 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). This section will present a brief survey of related works for this general problem.

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 [14] propose a 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 objects and processors) and topological variations (difference in shapes of the object graph and topology graph) are considered.

Local search techniques have also been proposed. 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 [16] 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. Bhanot et al [8] use a simulated annealing approach to optimize task layout on the BlueGene machine which works well for long running communication intensive applications.

Strategies for specific topologies and/or specific task graphs have also been studied. Ercal et al [7] provide a solution in the context of a 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 bi-partitioning is performed and the partition at the k^{th} iteration determines the k^{th} bit of the processor assignment. Bianchini and Shen [10] consider mesh network topology. Fang, Li and Ni [9] 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 the Hop-bytes metric we use (section 3), although considering only the communication with previously assigned tasks. A very similar scheme has also been implemented in CHARM++ (section 4.4). Taura and Chien [18] 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 (Object) Graph The parallel application is represented as a weighted undirected graph $G_t = (V_t, E_t)$. The vertices in V_t represent tasks (or objects) and the edges in E_t represent direct communication between the tasks (or 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 (object) in the vertex represent(s). 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 tasks (or objects) represented by v_a and v_b.

Task Mapping The task-mapping is represented by a map:

$$M : V_t \longrightarrow V_p$$

If the objects represented by the vertex $v_t \in V_t$ of the task-graph are placed on processor v_p, then $M(v_t) = v_p$. 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 :

$$M : V_t \longrightarrow V_p \cup \{\perp\}$$

where $M(v_t) = \perp$ 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 $HB(G_t, G_p, M)$ the hop-bytes when the task graph G_t is mapped on the topology graph G_p, under the mapping M. Alternatively, the overall Hop-bytes is half the sum of Hop-bytes due to individual nodes in the task graph.

$$HB(G_t, G_p, M) = \sum_{e_{ab} \in E_t} HB(e_{ab}) = \frac{1}{2} \sum_{v_a \in V_t} HB(v_a)$$

$$where \quad HB(e_{ab}) = c_{ab} \times d_p(M(v_a), M(v_b)) \quad and$$

$$HB(v_a) = \sum_{v_b | (v_a, v_b) \in E_t} c_{ab} \times d_p(M(v_a), M(v_b))$$

Hops per byte This is the average number of network links a byte has to travel under a task mapping.

$$Hops\ per\ Byte = \frac{HB(G_t, G_p, M)}{\sum_{e_{ab} \in E_t} c_{ab}}$$

$$Hops\ per\ Byte = \frac{\sum_{e_{ab} \in E_t} c_{ab} \times d_p(M(v_a), M(v_b))}{\sum_{e_{ab} \in E_t} c_{ab}}$$

4 The mapping heuristic

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 tasks. A partitioning method that reduces inter-task communication by placing heavily communicating objects in the same task must be preferred. In the next phase, the *mapping phase*, the p tasks are mapped onto the p processors such that more heavily communicating tasks 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. 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 becomes more complex due to the additional constraint of balancing the compute load on processors. We have adopted the two-phased approach in this paper.

The partitioning in the first phase is accomplished either using METIS [13] or using some of the existing topology-oblivious load balancing strategies in CHARM++. The mapping phase uses the following algorithm.

4.1 The algorithm

We employ an iterative approach in which the main question 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 current cycle. The estimation function has the following form: $f_{est}(t, p, M) \longrightarrow value$ where t

is an unassigned task, p is an available processor and M 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 some tasks it may not matter much if they are placed on their 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 current 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 current cycle. We then select the most critical task for placement in the current cycle.

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: $M : V_t \longrightarrow V_p$ (A task mapping)

  $T_1 \longleftarrow V_t$;
  $P_1 \longleftarrow V_p$;
  for $k \leftarrow 1$ ***to*** n **do**
    //Select the next task and processor (t_k, p_k);
    //Task t_k, is the one with max. criticality;
    $max\_criticality \leftarrow -\infty$;
    for *task* $t \in T_k$ **do**
      $criticality(t) = \frac{\sum_{p \in P_k} f_{est}(t,p)}{n-k} - \min_{p \in P_k} f_{est}(t,p)$;
      if $criticality(t) > max\_criticality$ **then**
        $t_k \leftarrow t$;
        $max\_criticality \leftarrow criticality(t)$;
      end
    //Processor p_k, is the one where t_k costs least;
    $min\_cost \leftarrow \infty$;
    for *processor* $p \in P_k$ **do**
      if $f_{est}(t_k, p) < min\_cost$ **then**
        $p_k \leftarrow p$;
        $min\_cost \leftarrow f_{est}(t_k, p)$
      end
    $M(t_k) = p_k$;
    $T_{k+1} \leftarrow T_k - \{t_k\}$;
    $P_{k+1} \leftarrow P_k - \{p_k\}$;
end

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 criticality for each task if it is placed in the current cycle. The estimation function should be such that $f_{est}(t, p, M)$ approximates the contribution of task t (if placed on processor p) to overall quality of the mapping. The function is topology-sensitive. Once criticality values are known for each task, the one with maximum criticality is selected. It is mapped to the processor where f_{est} estimates it to cost the least.

4.2 Estimation functions

In this section we will motivate and present multiple cost estimation functions. The function is used for estimating 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 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.

$$HB(G_t, G_p, M) = \frac{1}{2} \sum_{t_i \in V_t} HB(t_i),$$

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

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 $P_k \cup \bar{P_k} = V_p$.

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:

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

This is quite cheap to compute as compared to the other approximations. This estimation function has been used in the mapping strategy described in 4.4.

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 $M(t_j)$ by the *expected* distance of p to other processors. The distribution of $M(t_j)$ is taken to be uniformly random on P_k. In other words, for any unmapped task $t_j \in T_k$ we approximate:

$$d_p(p, M(t_j)) \approx \frac{\sum_{p_j \in V_p} d_p(p, p_j)}{|V_p|}$$

Thus we can refine our estimation function to be:

$$f_{est}(t_i, p, M) = \sum_{t_j \in \bar{T}_k} c_{ij} d_p(p, M(t_j)) + \sum_{t_j \in T_k} c_{ij} \frac{\sum_{p_j \in V_p} d_p(p, p_j)}{|V_p|}$$

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 $M(t_j)$ to be uniformly random on *available* processors P_k. In other words, for any unmapped task $t_j \in T_k$ we approximate:

$$d_p(p, M(t_j)) \approx \frac{\sum_{p_j \in P_k} d_p(p, p_j)}{|P_k|}$$

Since the consideration of running time dominates in the real-world applications, we will use first and second order approximation schemes in our implementation and results. This will be discussed in section 4.3.

4.3 Implementation: TopoLB2

The mapping algorithm with second order approximation has been implemented in CHARM++ as a strategy called TopoLB2 under the dynamic load-balancing framework. Initially, the object graph is partitioned into p tasks 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 $f_{est}(t, p, M)$. Rows are indexed by task nodes and columns are indexed by processors. The entry in the cell (t, p) is the current value of $f_{est}(t, p, M)$. 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 $f_{est}(t_k, p, M)$ 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 RT_{II} over all p iterations is:

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

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 $f_{est}(t, p, M)$ 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 $O(p)+O(p^2) = O(p^2)$. Thus, overall running time RT_{III} is:

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

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, due to small constant degree of the nodes of the task graph, 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 appproximation. Scaling considerations lead us to the choice of second order approximation for our scheme.

4.4 TopoLB1: A simpler strategy

TopoLB1 is a simpler load balancing strategy for CHARM++ which uses first order approximation scheme of estimation functions (section 4.2). The placement is most critical for a task that has maximum communication with already assigned tasks. This simplifies the task selection loop of algorithm 1. The algorithm is implemented using a 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 RT_I is:

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

Thus, the running time for first order approximation (RT_I) and for second order approximation (RT_{II} : see section 4.3) are asymptotically similar, though the constant factor is higher for TopoLB2. Hence, a more informed decision is made without any additional asymptotic cost. A similar strategy has been described by T. Baba et.al. [3]; where TopoLB1 corresponds to the (P_3, P_4) scheme.

5 Experiments

We now discuss and compare the performance of the load balancing schemes described earlier to a load balancer which places the tasks on the processors at random. Section 5.2 will describe the performance of TopoLB2 in reducing the hops-per-byte metric in different scenarios. The effect of this reduction on average message latencies in the network is analyzed via detailed simulations in section 5.3. Performance results on BlueGene/L [1] are presented in section 5.4.

5.1 Evaluation mechanism

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. 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. The load balancing framework automatically instruments all CHARM++ objects, collects computation and communication load during execution and stores it into a load balancing database. Based on the information from the database, the load balancing strategies decide on a new mapping of objects to processors. This works well when the computational loads and communication patterns of applications tend to persist over time, which is the case for many parallel applications, especially physical simulations which are iterative in nature.

The CHARM++ load balancing framework allows the runtime to log load information from an actual parallel execution into a file for off-line analysis. The effect of different 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 and specifying the log file and the load balancing step. 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 (TopoLB2) 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 object graphs that have just p objects (equal to the number of processors) 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 objects (or tasks) which communicate in a 2D-Mesh pattern. Each object communicates with its four neighbors (three or two for boundary and corner objects, respectively) in each iteration. The number of objects 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 processors and gives us the flexibility of connecting the processors in many different topologies.

5.2.1 2D-Mesh pattern on 2D-Torus

Figure 1 compares the performance of random placement, TopoLB2 and TopoLB1 in mapping a 2D-Mesh pattern onto a 2D-Torus topology. 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.

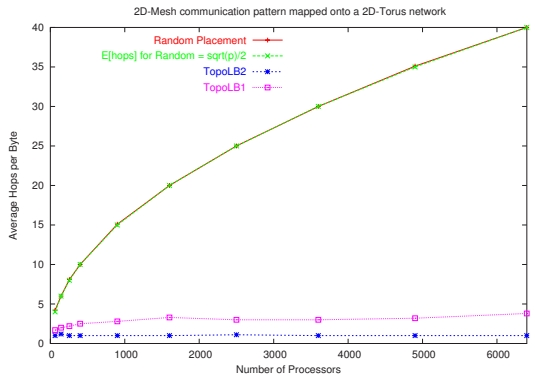

Figure 1: Mapping 2D-Mesh communication pattern onto a 2d-Torus. Random placement matches expected value.

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 TopoLB2 actually produces an optimal mapping in most cases. Figure 2 shows the comparison of TopoLB2 and TopoLB1 and is essentially a zoomed-in version of figure 1. It is also seen that TopoLB1 also results in small values of hops-per-byte, though TopoLB2 produces a better map than TopoLB1 in all tested cases.

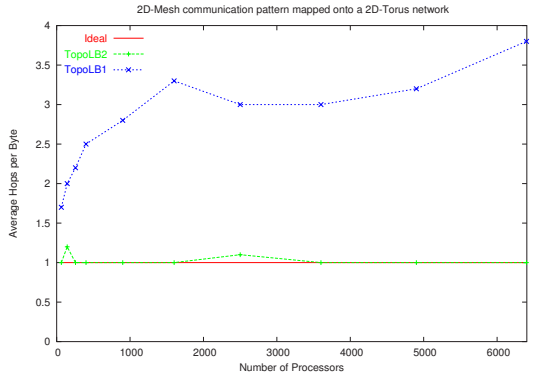

Figure 2: Mapping 2D-Mesh communication pattern onto a 2d-Torus. Zoomed in to compare TopoLB2 and TopoLB1.

The comparison of the time taken by the mapping strategies is shown in Table 2. The experiments were conducted on an Intel Pentium 4 machine running at 3.00 GHz with 1GB memory and 512KB cache. We can see that the decision-making times for both the strategies are comparable.

5.2.2 2D-Mesh pattern on 3D-Torus

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

Processors	Time for TopoLB2	Time for TopoLB1
64	2.24ms	1.99ms
256	37.96ms	29.32ms
400	99.68ms	68.26ms
900	551.28ms	320.07ms
1600	2.03s	0.98s
2500	5.30s	2.37s
3600	10.86s	4.88s
4900	24.14s	9.34s
6400	47.47s	15.99s

Table 2: Comparison of time taken in mapping a 2D-Mesh pattern on a 2D-Torus topology

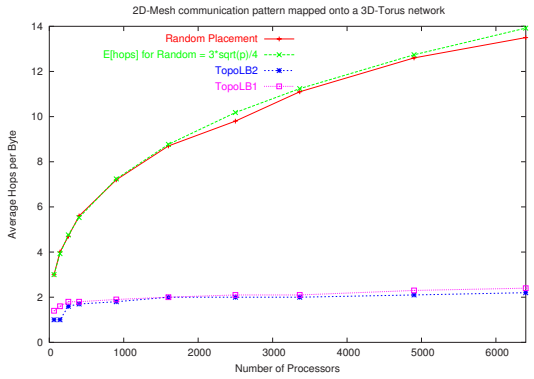

Figure 3: Mapping 2D-Mesh communication pattern onto a 3d-Torus. Random placement matches expected value.

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, TopoLB2 and TopoLB1, 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 mapping 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 3 that in this case, TopoLB2 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, TopoLB2 leads to a small value of hops-per-byte. TopoLB1 also results in small values of hops-per-byte which are about 10% higher than those from TopoLB2.

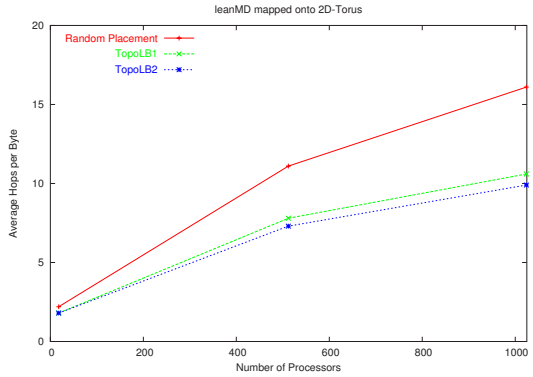

Figure 4: Comparison of different mapping strategies on 2D-tori for LeanMD data

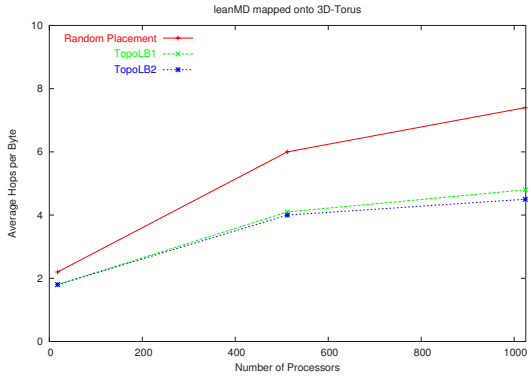

Figure 5: Comparison of different mapping strategies on 3D-tori for LeanMD data

5.2.3 LeanMD mapped onto different topologies

This section will describe the results of mapping communication pattern from a real molecular dynamics simulation program called LeanMD [15]. We have load information dumps for LeanMD on different numbers of processors. The total number of objects 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 objects is greater than the number of processors, we need to perform clustering of objects into p tasks with balanced computation load. We use METIS for this initial grouping.

Figure 4 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 objects in each task, almost all pairs of tasks communicate with each other. The average degree of the coalesced task-graph obtained from METIS is 12.7, which means that each task communicates with 70% of the tasks. Hence it is difficult for any strategy to reduce hop-bytes as almost all the tasks communicate. For 512 processors, the virtualization ratio is 6 and the average degree of the coalesced task graph is 19.5 which means that each task communicates with about 4% of the other tasks. This creates some avenues for intelligent placement of tasks to keep the communication local. As seen from figure 4, TopoLB2 leads to a 34% reduction in average hops-per-byte over random placement on 512 processors. TopoLB1 also performs well, leading to a 30% reduction; similar trend is seen for 1024 processors.

Figure 5 shows the results for mapping onto 3D-Tori. The relative performance of the different schemes in this case is similar to the last case.

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 and other characteristics on the network.

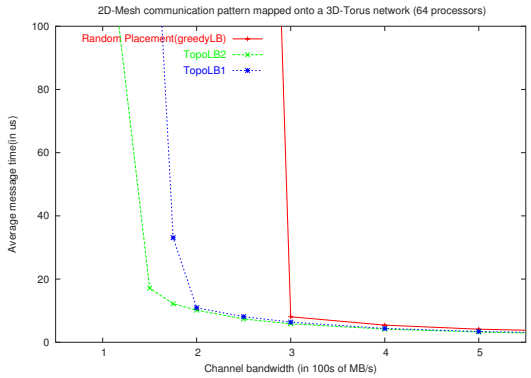

Figure 6: 2D-mesh on 64-node 3D-Torus: Average message latency using different mappings

We performed simulations using BigNetSim [20], 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 object 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 TopoLB2, TopoLB1, 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.

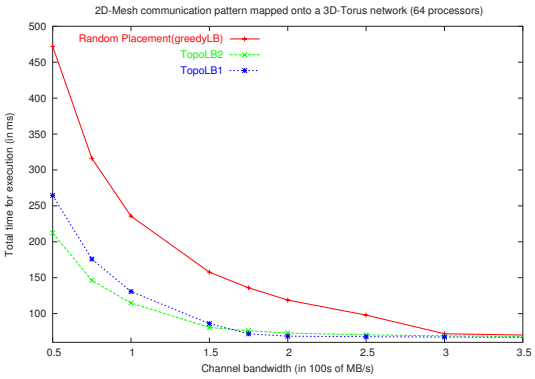

Figure 7: Completion time for the execution of 2000 iterations

The execution of application traces is simulated on a (4,4,4)3D-Torus interconnection network. A smaller network is chosen due to fine-grained computation of the objects and time taken by the network simulator to run the application traces. Since TopoLB2 and TopoLB1 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 6 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. TopoLB1 can tolerate a further reduction in network bandwidth while TopoLB2 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.

The total time for the entire execution to finish is also improved by using intelligent mapping. Figure 7 shows the total time required for the completion of 2000 iterations of the benchmark. For smaller bandwidth, optimizations obtained by TopoLB2 and TopoLB1 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 TopoLB2. TopoLB1 also leads to a large reduction over random placement. However, TopoLB2 outperforms TopoLB1 by about 10-25%.

5.4 Results on BlueGene/L

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

Figures 8 and 9 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 TopoLB2 and TopoLB1 lead to reductions in time when compared to random mapping. Note that the number of objects 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.

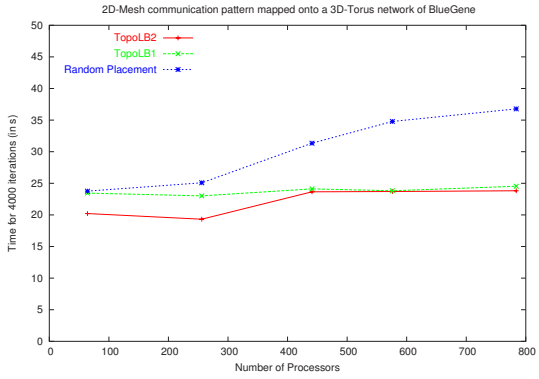

Figure 8: Comparison of mapping strategies on BG/L 3D-Torus network.

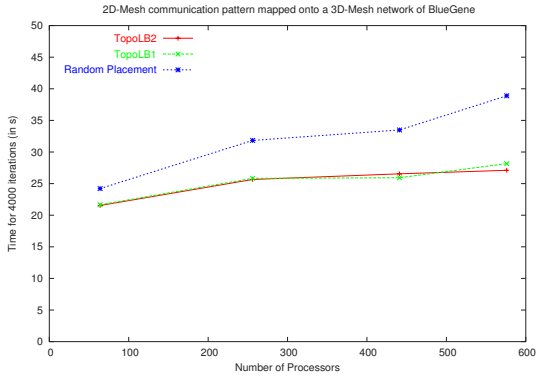

Figure 9: Comparison of mapping strategies on BG/L 3D-Mesh network.

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 TopoLB2 and TopoLB1 map 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 algorithm that 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 TopoLB2 provides a good mapping in terms of average number of hops travelled by each byte, and compares favorably with some other schemes. We also developed another similar, but simpler and faster, scheme called TopoLB1 for the purpose of comparison of its results with TopoLB2. 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.

Detailed simulations (section 5.3) and real-machine studies (section 5.4) serve distinct purposes in this work. With simulation, one can vary physical parameters (such as channel bandwidth), but one is limited to smaller configuration due to the time complexity of detailed simulation. In contrast, real-machine studies allow us to demonstrate utility in realistic settings. In future, we hope to use parallel simulation to bridge the gap and validate the predictive power of the simulations.

Due to the massively large sizes of machines like BlueGene, a distributed approach toward keeping communication localized in a neighborhood may be needed for scalability in the future. Hybrid approaches, such as that in [19], 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] G. B. et al. Optimizing task layout on the bluegene/l supercomputer. *IBM Journal of Research and Development*, 49(2/3):489, 2005.

[9] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.