STRATEGIES FOR TOPOLOGY-AWARE TASK MAPPING AND FOR REBALANCING WITH BOUNDED MIGRATIONS

BY

TARUN AGARWAL

B.Tech., Indian Institute of Technology, Delhi, 2003

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

The efficient usage of parallel machines requires that both the compute nodes as well as the interconnection network be utilized efficiently. As the number of processors in parallel machines increases, the diameter of the interconnection topology can become quite large. In order to prevent messages from travelling over such large distances, the entities in a parallel application need to be mapped onto the interconnection topology so that communicating objects are placed in the same neighborhood. This thesis presents a mapping scheme, which produces good mappings that reduce the average number of hops travelled by a byte on the network. We find that a mapping produced by our scheme reduces the effective cost of communication on the network since packets encumber a fewer number of links on an average. This thesis also studies the problem of load rebalancing that arises in the context of dynamic load balancing. After the initial mapping, further load balancing steps can reduce the migration cost by adjusting the current mapping by migrating only a few objects. We compare two schemes for this problem, and show that a simple greedy heuristic works well in practice.

To my grandfather

Acknowledgments

I would like to thank Prof. L. V. Kale who advised me on this thesis. His guidance, advice and encouragement have been invaluable over the last two years, and particularly for the work in this thesis.

I also thank Amit Sharma, my colleague at the Parallel Programming Laboratory, with whom I worked closely during this work. His high energy level and cooperative nature made this work easier. Gengbin Zheng helped with his expertise in all matters related to Charm++ load balancing whenever I had a question. His help is much appreciated. I am thankful to Nilesh Choudhury for his help with issues related to the network simulator used in this work. I have had a very enjoyable stay at the Parallel Programming Laboratory, for which I thank everyone in the lab.

Finally, I would like to thank my parents and my sister for their unconditional support that forms the basis of my strength. I remember my grandparents with a deep sense of gratitude for moulding me into what I am. Both of them passed away in the last two years.

Table of Contents

List of Tables

List of Figures

Chapter 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 [2] once fully deployed.

It is imperative that techniques for efficient utilization of these large scale machines be developed. A major challenge in scaling applications to large machines is to utilize resources uniformly. The main resources in a large parallel machine are its compute nodes and the interconnection network; both must be utilized efficiently. The virtualization model of Charm++ provides a good platform for exploration of ideas towards this end. 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 [18]. 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++. The benefits can be extended to AMPI (adaptive MPI), which is based on Charm++.

With a massively large number of processors, packaging considerations sometimes lead to the choice of a Mesh or a Torus topology. For example, the primary network in BG/L is a 3D-Torus. Even for a relatively moderate machine size a message might travel a large number of hops. 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 is also quite high. If packets travel over such large number of hops, the average load on the links increases, which increases contention. Therefore, it is desirable to map communicating objects to nearby processors. Some mapping strategies addressing this problem are discussed in this thesis.

Dynamic load balancing has an associated overhead of task migration. In Charm++ this is handled using the PUP framework [14] which is a way of describing the layout of object's data in memory. If every call to the load balancer leads to a new ab-initio mapping, a large fraction of objects migrate and the migration cost is high. Instead of starting from scratch, a load *rebalancing* approach can be applied which would involve the migration of a limited number of objects; one such scheme is discussed in this thesis.

1.1 Thesis contribution

The main contributions of this thesis are:

- *Mapping strategies:* A scheme for mapping task graphs onto arbitrary processor topology to reduce communication cost is designed and implemented. In addition, a simple improvement scheme that starts with a given mapping and only reduces the implied communication cost is implemented. These schemes have been implemented as TopoLB and RefineTopoLB load balancing strategies for Charm++ and AMPI amd are therefore available to all the applications developed using them.

- *Bounded migration:* A strategy that adjusts a task mapping to re-balance processor loads while migrating a bounded fraction of objects is implemented as a load balancer for Charm++, called RefineKLB. It is largely based on the scheme by G. Agarwal et.al.[3], with modifications to account for un-migratable objects and some heuristic improvements.

1.2 Thesis organization

Chapter 2 describes the basic Charm++ model and the dynamic load balancing framework. It also talks about some classifications of load balancers. The task mapping algorithms are described in 3. An overview of related work is also presented in this chapter. The chapter presents the following mapping strategies: TopoLB, with its variants, and RefineTopoLB. Performance results for the topology-aware schemes are presented in Chapter 4. Chapter 5 talks about the problem of load rebalancing through bounded migration. The chapter describes RefineKLB and practical issues related to its use. Experimental results are also given in this chapter. Finally, the thesis is summarized in Chapter 6, where some directions for future research are also given.

Chapter 2

Charm++ and processor virtualization

Charm++ [17] is a parallel programming environment that has the idea of processor *virtualization* at its core. The programmer breaks down her application into parallel objects depending on the nature of the application. The decomposition is performed independent of the available number of processors. Typically the number of objects, N, is much larger than the number of processors, P. These compute objects can be thought of as virtual processors. In Charm++ terminology they are called *chares*. Thus, the programmer only performs *problem decomposition* in the application domain, while the actual *task mapping* is done by the runtime system(RTS) [18]. This delegation of a part of the effort to the runtime system not only improves a programmer's productivity [19], but also creates avenues for automatic performance optimizations and features like fault tolerance [29].

2.1 The Charm++ programming model

In the programmer's view, a charm++ application consists of compute objects and their interaction. The compute objects are C++ objects, and the system allows them to interact via asynchronous method invocation on each other. Figure 2.1 shows the user's view and the system implementation of a Charm++ application.

Charm++ has a *message-driven execution* model. A non-preemptive scheduler executes

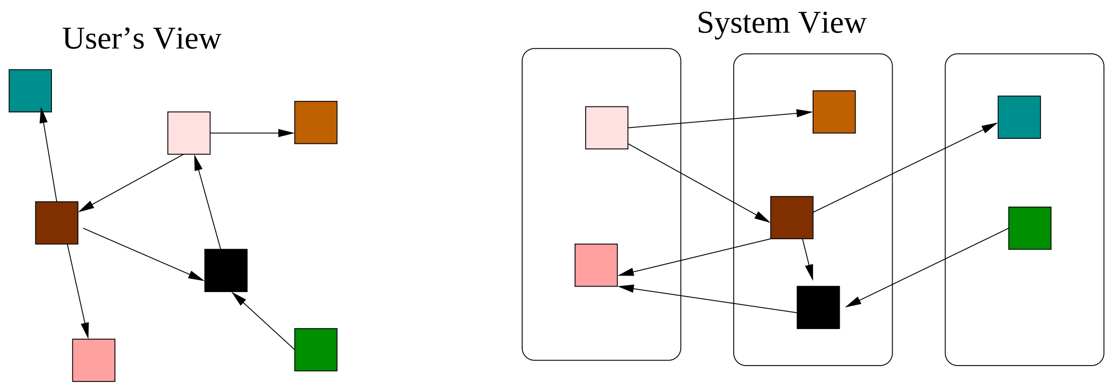


Figure 2.1: Virtualization: System takes care of actual mapping to physical processors

methods on objects as messages arrive. While one object is waiting for a message, another one can be executed on the same physical processor; this leads to an effective overlap of communication and computation.

2.2 Load balancing

The user code is independent of the actual mapping of compute objects to physical processors. This enables the Charm++ runtime to dynamically map and remap them. It can *migrate* objects from one processor to another during execution. The runtime ensures correct message delivery and collective operations, like reductions and broadcasts, in the presence of migrations.

One of the possibilities that migration creates for the runtime is to provide for automatic load-balancing. Objects can be migrated away from overloaded processors to under-loaded processors, this allows for *measurement-based* load balancing.

2.2.1 Static vs. Dynamic load balancing

A large number of parallel applications have irregular computational structure, which creates load imbalance across processors unless special care is taken. One way to perform load balancing is to model the application structure (through characterizing runs, etc.), then statically partition the application and map the partitions accordingly. However, such characterization is often difficult because the behavior changes with program input and environmental conditions. Moreover, in applications like fracture simulation the structure can change dynamically, making the static approach ineffective. The other approach is for the runtime to manage the partitioning and assignment dynamically. Charm++ employs this approach in its measurement-based load balancing framework. This approach works for scenarios where the *principle of persistence* applies. If the application's load and/or communication pattern shows temporal correlation, recent history can be used to predict the future. Many scientific applications, like physical simulations, fall into this category, they usually have an iterative structure and the computation lasts for a long time. This allows the runtime to create a dynamic model of the application through measurement during actual execution. Such a model (essentially a task-graph) is used to perform load balancing dynamically. This approach is called *measurement-based* load balancing.

2.2.2 Dimensions of load balancing

The problem of load balancing in the general form involves balancing computation load over the available physical processors, as well as minimizing communication across the network. However, different load balancing algorithms have been developed solving partial problems or using only some of the available degrees of freedom. They can be broadly put into the following categories:

1. *Computation load only*: This category contains partitioning algorithm. This problem is also known as *Makespan Minimization* and is known to be NP-hard. However, several heuristics have been developed here.

2. *Computation load and communication (topology-oblivious)*: Here, although communication is also considered, the actual network topology is not taken into consideration. Thus the objective is to balance computation load on processors while keeping most of

the communication intra-processor. Thus distinction is made only between intra and inter-processor communication, not between long-range and short-range inter-processor communication. This is like solving for a complete $N \times N$ network where every pair of processors is directly connected.

3. *Computation load and communication (topology-aware)*: Here the underlying topology is also considered. The objective is to balance computation load while keeping inter-processor communication short-range by placing communicating partitions on nearby processors.

The Charm++ system provides load balancers in all of the categories mentioned above.

2.2.3 Centralized vs. Distributed load balancing

Dynamic load balancing algorithms can be classified into two broad categories: *Centralized* and *Distributed*. In centralized load balancing, the entire load information is sent to a central processor which uses the global view to make load balancing decisions. For small machine sizes the load information is relatively small and a centralized strategy works fine. In larger machines a fixed node can be dedicated to performing load balancing. The other contrasting approach is a fully distributed approach, where load information is only exchanged with the neighborhood (according to either the actual topology or a virtual embedding). Distributed algorithms have been proposed especially for non-iterative tasks. Examples of distributed strategies include Adaptive Contracting Within Neighborhood (ACWN) scheme [15, 30] for the Chare (early Charm++) kernel, sender/receiver initiated diffusion based schemes [33], etc. However, for iterative applications distributed algorithms have a slow rate of convergence towards the global optimum.

Hybrid strategy

For a large machine centralized strategies have a bottleneck at the dedicated processor while distributed strategies suffer from slow rates of convergence. While centralized strategies may still be feasible in some cases through a dedicated central node and a dedicated network for load balancing information exchange, a new approach is needed. A *hybrid* approach can be followed [34] where nodes are clustered hierarchically into groups. The root node only performs load balancing across clusters, which requires much less resources at the root as well as less data transmission over the network. Each cluster balances itself independently, which makes the scheme distributed.

The Charm++ system provides centralized, distributed, as well as hybrid load balancing strategies for the user.

2.3 Dynamic load balancing framework in Charm++

One of the major objectives of the Charm++ system is to provide for automatic migration of parallel application with minimal effort from the user. The load balancing framework is designed in a way to make the selection of a load balancing strategy orthogonal to the actual migration mechanism. The decision making module of the load balancer is clearly separated from user code. The framework instruments the actual object load and communication information which is represented as a task graph. The decision-making module essentially operates on this application-independent task-graph representation. The initial basic version of the load balancing framework was implemented by Robert Brunner [16], which has since been redesigned and improved by Gengbin Zheng [34] and others to provide for richer functionality and incorporation of newer constructs in Charm++.

The various components of the load balancing framework are shown in figure 2.2. The figure shows the organization on a single processor. At the top level reside actual decision making modules, the load-balancing *strategies*. Strategies are implemented as Charm++

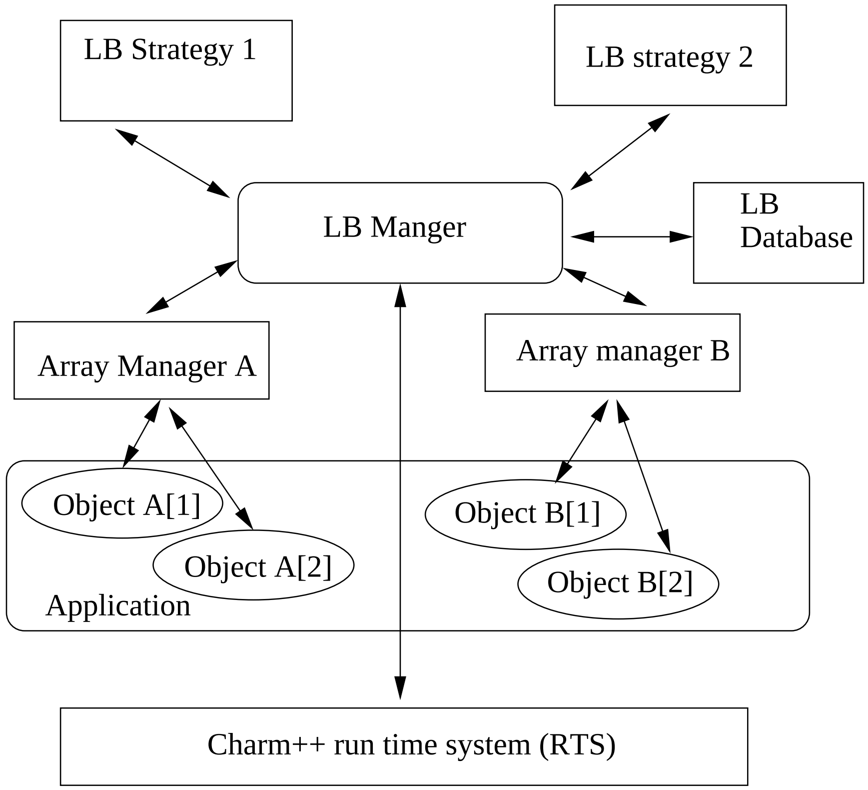


Figure 2.2: Components of Charm++ load balancing framework

Chare groups. A Chare Group is a collection of chares with one representative (group member) on each physical processor. The strategies are invoked by the *LB Manager*. The LB Manager is responsible for recording the times of execution of chares, their communication with other chares (both point to point and collective), background load on processors and idle times. When LB Manager asks the strategy to perform a load balancing decision, the strategy can get the information about local objects, their execution times, and communication patterns from the local *LB Database*. The strategy can communicate with other processors to get load information on them. While distributed strategies take information from processors in the neighborhood, in centralized strategies information is sent to a central processor (usually processor 0). With detailed information about background load, object computation load, and communication pattern, the strategy module makes a decision about new assignment of objects.

The LB Manager supervises the migration of objects via *Array Managers*. A *Chare*

Array is a collection of chares where each member has global unique index where messages are sent. There is an Array manager corresponding to each Chare Array. Array Managers inform the LB Manager about the arrival and departure of their chares. They are also responsible for informing the LB Manger about the starting of the execution of their chares, so that it can record execution times for individual chares. Similarly, when communication is initiated by a chare, the corresponding Array Manager informs the LB Manager to record it in the database. The LB Manager makes the database available to the strategies which make migration decisions. The LB Manager informs the Array Managers to perform the actual migration.

This thesis presents the author's work on new strategies for load balancing. Chapter 3 presents efforts for optimizing communication locality through topology-aware load balancing. In addition, strategies to *re-balance* load by migrating only a few (bounded) chares are presented in chapter 5.

Chapter 3

Topology-aware load balancer

The problem of load-balancing is two-fold. At the basic level, the compute load across the processors must be balanced. This is because the processor with maximum load prevents others from progressing. At the next level, communication over the links must ideally be contention-free; this is because contention for the network also prevents the contenders from progressing. Thus, contention for the links in the network must also be reduced.

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 more 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 take the two-phased approach in this chapter.

3.1 Types of topologies

- **Mesh**

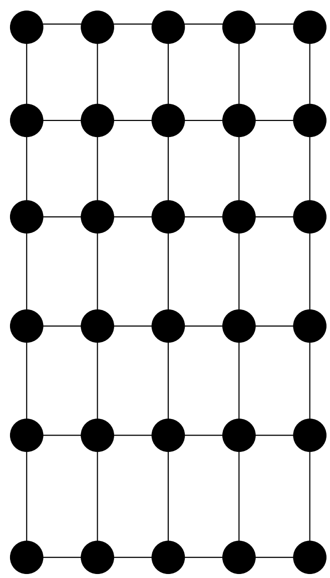


Figure 3.1: A (6,5) 2D-Mesh Topology

Mesh topology is also known as a Grid topology. An $(a_1, \ldots, a_n)$ n-dimensional Mesh consists of processors placed on integer coordinates of an n-dimensional coordinate system in the space $[(0, \ldots, 0), (a_1, \ldots a_n))$. Processors that differ in exactly one coordinate by 1 are connected. For example, a 1D Mesh is just a linear arrangement and a $(2, 2)$ 2D Mesh is a square. Figure 3.1 shows a $(6, 5)$ 2D-Mesh. The number of connections per processor depends only on the dimension and is independent of the total number of processors.

- **Torus**

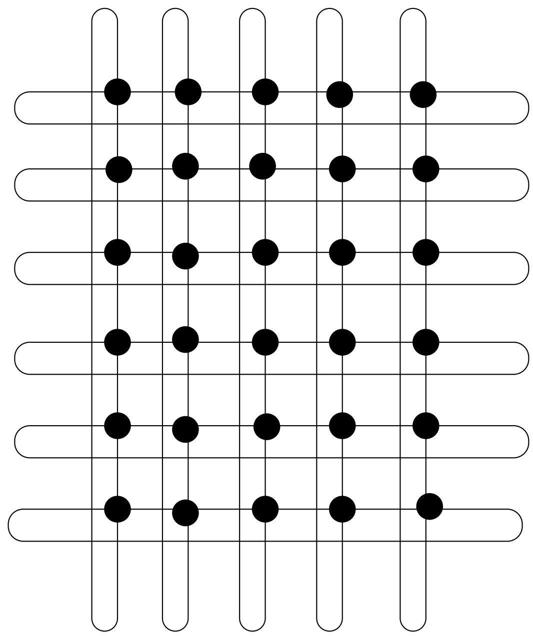


Figure 3.2: A (6,5) 2D-Torus Topology

A Torus is very similar to a Mesh but contains extra circular connections. An $(a_1, \ldots, a_n)$ n-dimensional Torus also consists of processors placed on integer coordinates of an n-dimensional coordinate system in the space $[(0, \ldots, 0), (a_1, \ldots a_n))$. A Torus contains all the connection that the corresponding Mesh contains. In addition, extreme processors in each dimension also have a circular connection. Formally, processors that differ in exactly one coordinate, say j^{th}, by 1 $mod(a_j)$ are connected. For example, figure 3.2 shows a $(6, 5)$ 2D-Torus.

- **Hypercube** An nD-hypercube is the generalization of a 3D-cube in n dimensions. It contains 2^n processors. An nD-hypercube is the same as as a $(2, \ldots, 2)$ nD-Mesh. Thus, all coordinates of a processor on a hypercube are either 0 or 1. In order to connect p processors in a hypercube we need to make a $\log(p)$D-hypercube. A hypercube doesn't

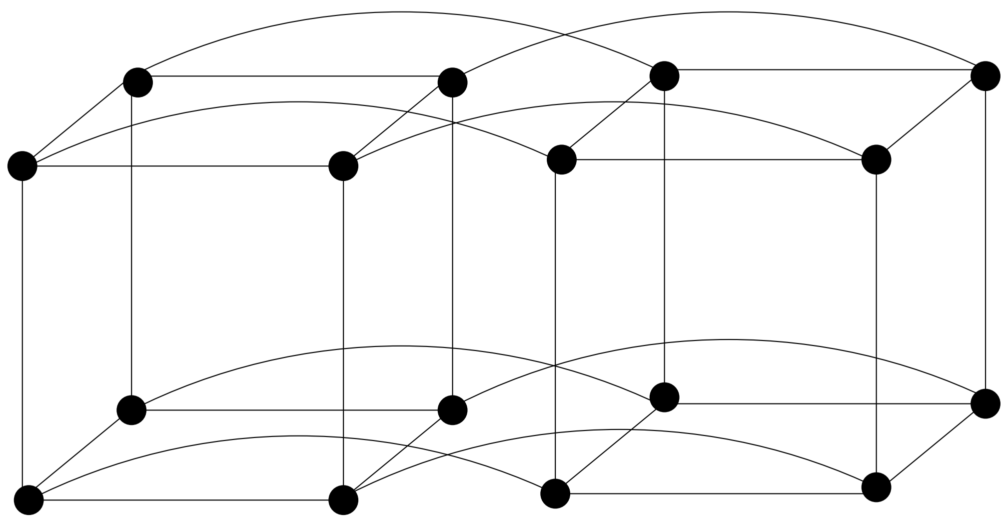


Figure 3.3: A 4D-Hypercube Topology

scale very well because the number of physical connections needed per processors is $\log(p)$, which quickly becomes large (for example 8 connections are needed for 256 processors).

3.2 Basic 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 the edges in E_p represent 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. The edges are undirected, which means that we are not considering the direction of communication in our algorithm.

- **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** 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 $v_{1p}, v_{2p} \in V_p$, the distance between them is represented by $d_p(v_{1p}, v_{2p})$. 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.

$$HB(G_t, G_p, P) = \sum_{v_a \in V_t} HB(v_a)$$

$$where\ HB(v_a) = \frac{1}{2} \sum_{e_{ab} \in E_t} HB(e_{ab})$$

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

$$Hops\ per\ Byte = \frac{HB}{\sum_{e_{ab} \in E_t} c_{ab}}$$

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

3.3 Related work

The problem of scheduling 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. 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 objects into balanced groups 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 3.2). This section will present a brief survey of related works in all these categories.

3.3.1 Communication-oblivious partitioning

The problem of assigning a set of n jobs (each with some arbitrary size) on p processors, so as to minimize the maximum load (makespan) on the processors is called the Makespan minimization problem and is well studied. It is an NP-hard optimization problem [25].

Hochbaum and Shmoys give a polynomial approximation scheme for uniform machines [12]. T.D.Braun et.al. [9] study a variety of mapping heuristics for heterogeneous machines. Min-min heuristic assigns the smallest overall job to its biggest machine, Max-min assigns the largest overall job to its best machine. Search techniques like genetic algorithms (GA), simulated annealing, and A* are also considered. It is observed that search techniques like GA give the best result but are quite slow [9]. For homogeneous systems, a simple greedy algorithm where jobs are assigned in decreasing order of sizes to the current least loaded machine performs well in practice. Though the algorithm can guarantee only a 2-approximation, it gives much better results with jobs being small (relative to average load on processors).

3.3.2 Communication-aware partitioning

In the context of parallel programs, a model where tasks *communicate* better reflects reality. While it is desirable to reduce makespan, performance is also effected by communication requirements. Intra-processor communication being relatively much faster, it is also desirable to reduce inter-processor communication. Stramm and Berman [31] formulate cost function that model processor loads and communication. They use these cost functions to guide local neighborhood search and simulated annealing search over the possible configuration space. It is observed that a good starting configuration is needed for these searches to lead to a good result. Mansour et.al. [26] present a graph contraction algorithm where highly communicating neighbors are repeatedly merged. The Kernighan-Lin algorthm [23] starts with an intial mapping and refines it by swapping pairs of objects across group boundaries, so that cross-edges are reduced. Recursive multilevel k-way partitioning is implemented in METIS [22].

3.3.3 Topology-aware mapping

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 [7] 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 [7] starts with an initial mapping and performs pairwise exchanges to improve the metric. Results are given for up to 49 tasks. Lee and Aggarwal [24] 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 [6] 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 [8] 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 [4] 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 [28, 1] 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 [10] 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 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 [13] consider mesh network topology. Fang, Li and Ni [11] study the problem of 2-D convolution on mesh, hypercube and shuffle-exchange topologies only.

Baba, Iwamoto and Yoshinaga [5] 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 [32] 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.

3.4 TopoLB

We now present a mapping algorithm that aims to reduce *hops per byte*. The basic structure will be presented and the different variations of the basic idea will be considered.

3.4.1 Intuition

Since we take the approach of placing objects one by one, 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. We will consider different estimation functions. 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.

3.4.2 The algorithm

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

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,P)) / (n−k) − min_{p∈P_k} f_est(t, p, 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, P) < min_cost then
                p_k ← p;
                min_cost ← f_est(t_k, p, P)
        end
        P(t_k) = p_k;
        T_{k+1} ← T_k − {t_k};
        P_{k+1} ← 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 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.

3.4.3 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.

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

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

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:

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

2. *Second order approximation*

   We will approximate the contribution of communication with taks 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:

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

Thus we can have a more accurate estimation function below:

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

While using a better approximation in the estimation function 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. However, the consideration of running time dominates here, and we will use the second order approximation scheme. This will be discussed in section 3.4.4.

3.4.4 Implementation of the algorithm

The load balancing algorithm has been implemented in Charm++ as a strategy called TopoLB under the framework described in section 2.3. Initially, the task graph is partitioned into p groups using METIS [20, 22, 21]. Any other topology-oblivious partitioner can also be specified for partitioning. At this point, both the new task graph and the topology graph have the same size p. 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.

- *First/Second order approximation*

  If we are using either the first order approximation or 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:

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

  While the running time $O(p|E_t|)$ can be as bad 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.

- *Third order approximation*

  In this case 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 takes the same time as the first order approximation ($O(p|E_t|)$) while third order approximation costs more ($O(p^3)$). 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 appproximation. Scaling considerations lead us to the choice of second order approximation for our scheme.

3.5 RefineTopoLB : A topology based refiner

RefineTopoLB is a topology-based refiner. It is intended to be used for further reducing communication cost (hop-bytes) after applying an initial load balancer like TopoLB. It essentially assumes that the compute load on processors is balanced. All chares assigned to the same processor are thought of as forming a chunk. The refiner swaps entire chunks between processors to see if more long-range communication is reduced. The relevant metrics here also is hop-bytes. Each chunk contributes to the overall value of hop-bytes as the sum of its communication with other chunks (say, in bytes) weighted by the distance between their assigned processors. The strategy starts with the most communicating chunk and tries to find if swapping its location with another chunk improves the overall hop-bytes. If an improvement is possible, the chunk swaps its processor with the best choice. Similarly, the strategy tries a swap for all chunks in order of their total communication. Since at each step a swap is performed only if there is an improvement, the final result is guaranteed to be better than the original assignment in terms of hop-bytes. The selection of best place for each swap takes $O(p^2)$; so the overall running time of the algorithm is $O(p^3)$. The high cost makes it usable only for small topologies.

3.6 TopoCentLB

TopoCentLB is a topology-aware load balancer for Charm++ developed by Amit Sharma. 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 each iteration, the task that has maximum total communication with already assigned task 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. A similar scheme has been described by T. Baba et.al. [5]; this scheme corresponds to their (P_3, P_4) scheme.

We will present a comparitive study of TopoLB and TopoCentLB in chapter 4.

Chapter 4

Performance results for TopoLB

In this chapter we will discuss the performance of the load balancing schemes described earlier. Section 4.2 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 4.3.

4.1 Evaluation mechanism in Charm++

Charm++ load balancing framework allows the runtime to dump 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 dumped as runtime parameters (using +LBDump *StartStep* to specify the first step, and +LBDumpSteps *NumSteps* to specify the total number of steps). A dump 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 dump 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 dump 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 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.

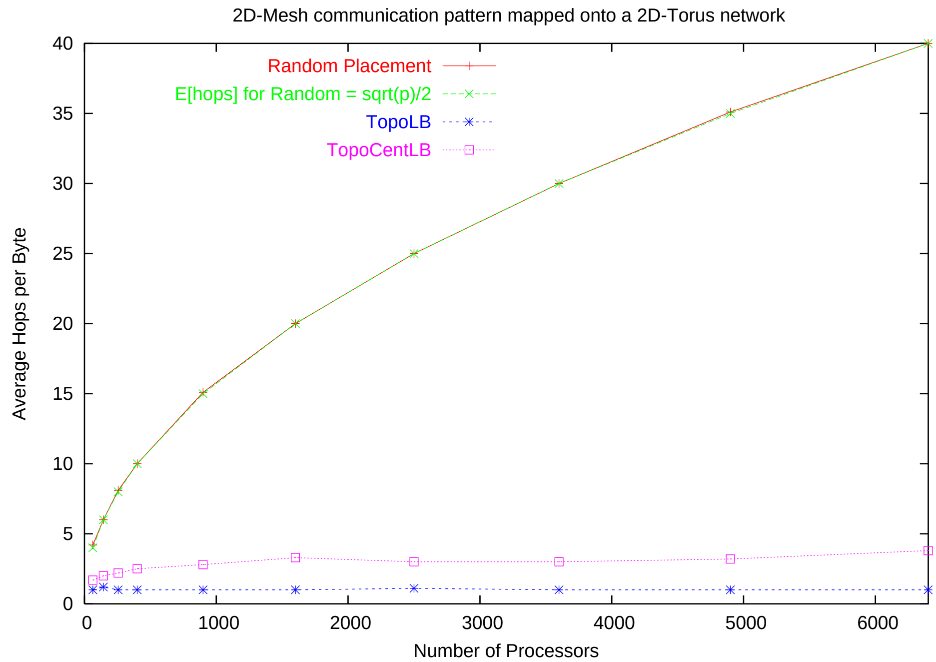
4.2 Reduction in hop-bytes

As described in chapter 3, the metric that TopoLB aims to reduce is hop-bytes, or equivalently, hops-per-byte. We will present TopoLB's performance in terms of hop-bytes reduction.

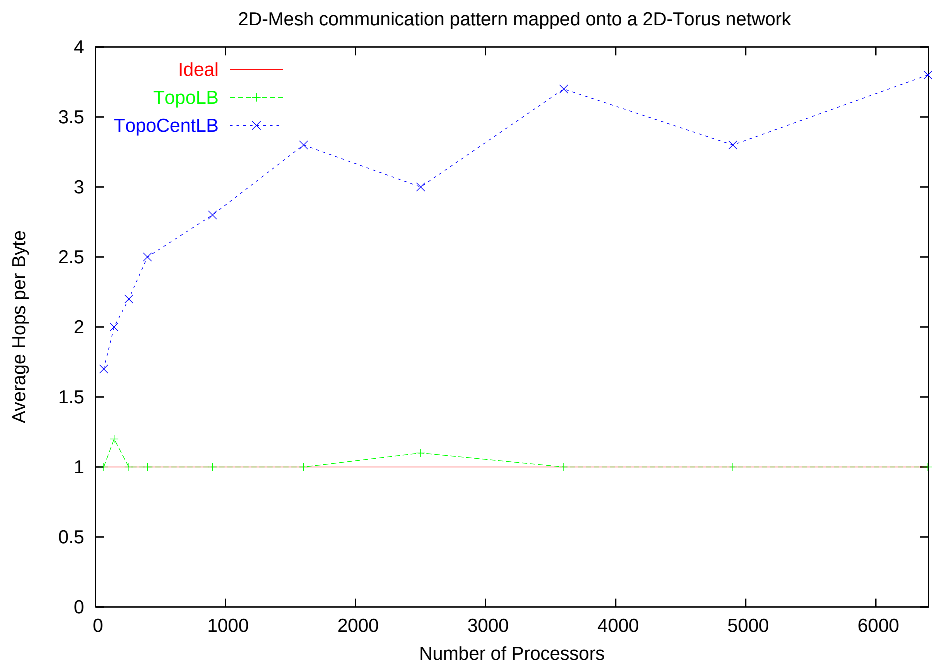
TopoLB assumes that the chares have been grouped into p groups, where p is the number of processors, and then maps the coalesced task graph onto the topology graph. To study the quality of mapping independent of the clustering method, we can start with task graphs that have just p chares so that no clustering is needed. We use a Charm++ benchmark program which has a jacobi-like communication pattern for this purpose. The benchmark program creates chares 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.

4.2.1 2D-Mesh pattern on 2D-Torus

Figure 4.1(a) 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. Since the placement is random, the expected number of hops a packet needs to travel is the expected distance between two ran-



(a) Random placement matches expected value



(b) Zoomed in view to compare TopoLB vs. TopoCentLB.

Figure 4.1: Mapping 2D-Mesh communication pattern onto a 2d-Torus. Number of chares is same as number of processors.

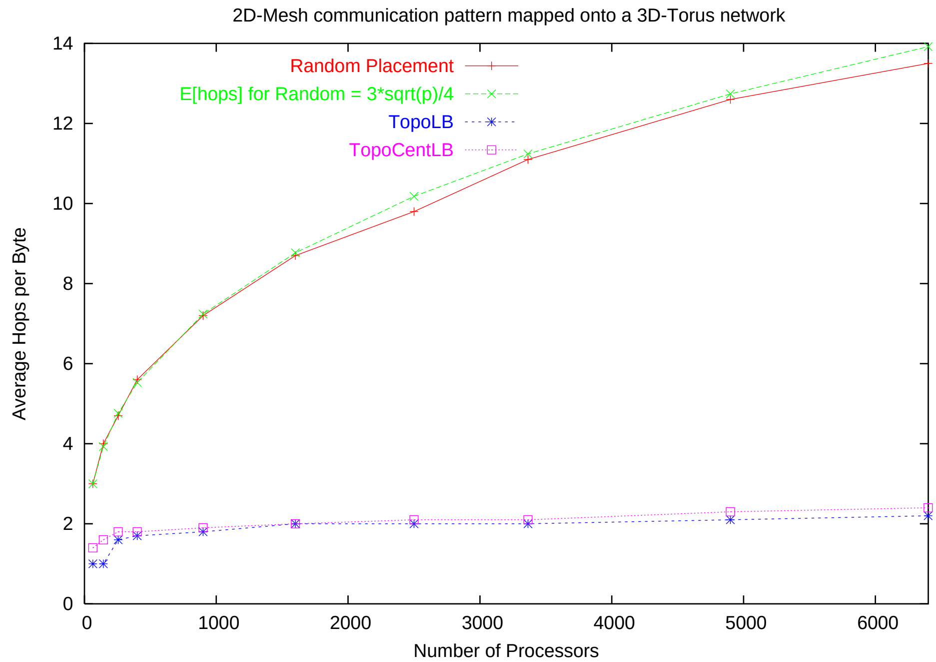
Number of processors	Time for TopoLB (in s)	Time for TopoCentLB (in s)
64	0.0022410	0.0019920
144	0.0130850	0.0097710
256	0.0379570	0.0293210
400	0.0996770	0.0682570
900	0.5512810	0.3200670
1600	2.0296010	0.9815290
2500	5.2971460	2.3748240
3600	10.8602030	4.8823460
4900	24.1431710	9.3374660
6400	47.4692040	15.9934400

Table 4.1: Comparison of time taken in mapping a 2D-Mesh pattern on a 2D-Torus topology

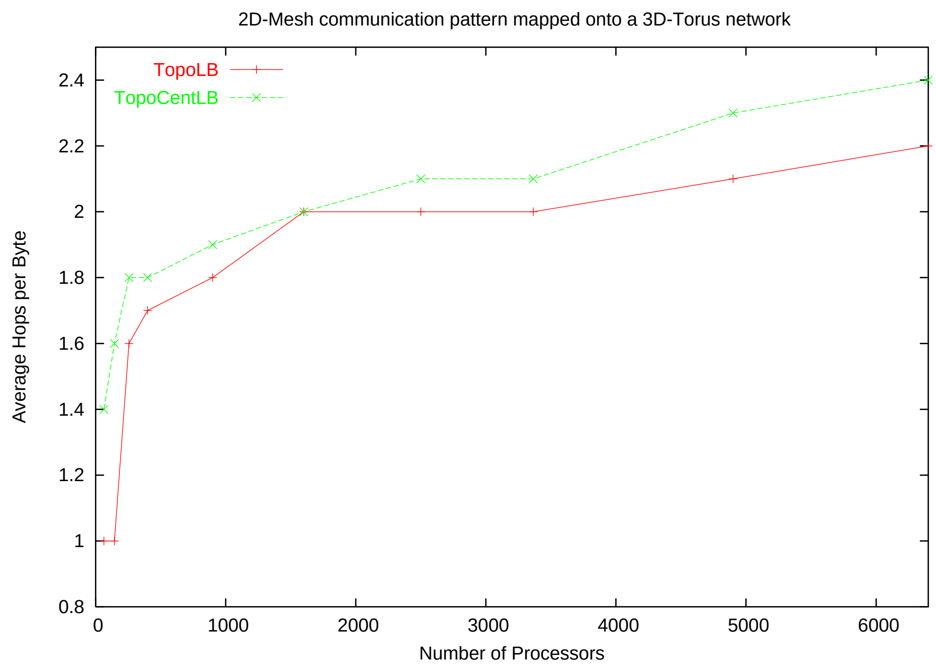
dom processors. The distance between two processors in a 2D-Torus is the sum of distances in the two dimensions. Each dimension has a span of $\sqrt{p}$, and with a wrap-around link the expected distance in each dimension is $\frac{\sqrt{p}}{4}$. Thus, the total expected distance between two random processors is $2\frac{\sqrt{p}}{4}$, or $\frac{\sqrt{p}}{2}$. As seen in figure 4.1(a), the value of hop-bytes for random placement matches closely with this 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 4.1(b) shows the comparison of TopoLB and TopoCentLB and is essentially a zoomed-in version of figure 4.1(a). 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.

The time taken by the mapping strategies in mapping a 2D-Mesh communication pattern onto a 2D-Torus of the same size is shown in Table 4.1. The experiments were conducted on an Intel Pentium 4 CPU machine running at 3.00 GHz with 1 GB of memory and 512KB cache. We can see that TopoCentLB scales better than TopoLB.



(a) Random placement matches expected value



(b) Zoomed in view to compare TopoLB vs. TopoCentLB.

Figure 4.2: Mapping 2D-Mesh communication pattern onto a 3d-Torus. Number of chares is same as number of processors.

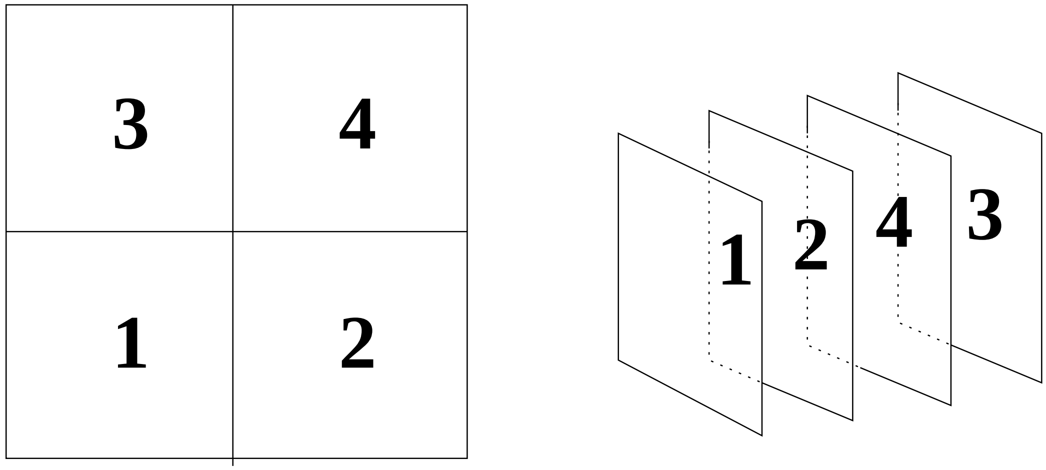


Figure 4.3: A (8,8)2D-Mesh mapped optimally onto a (4,4,4)3D-Torus

4.2.2 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 4.2(a). For a 3D-Torus, the expected distance between two random processors is $3\frac{\sqrt[3]{p}}{4}$. As seen in figure 4.2(a), 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 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. One such mapping is shown in figure 4.3. We can see from figure 4.2(b) 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.

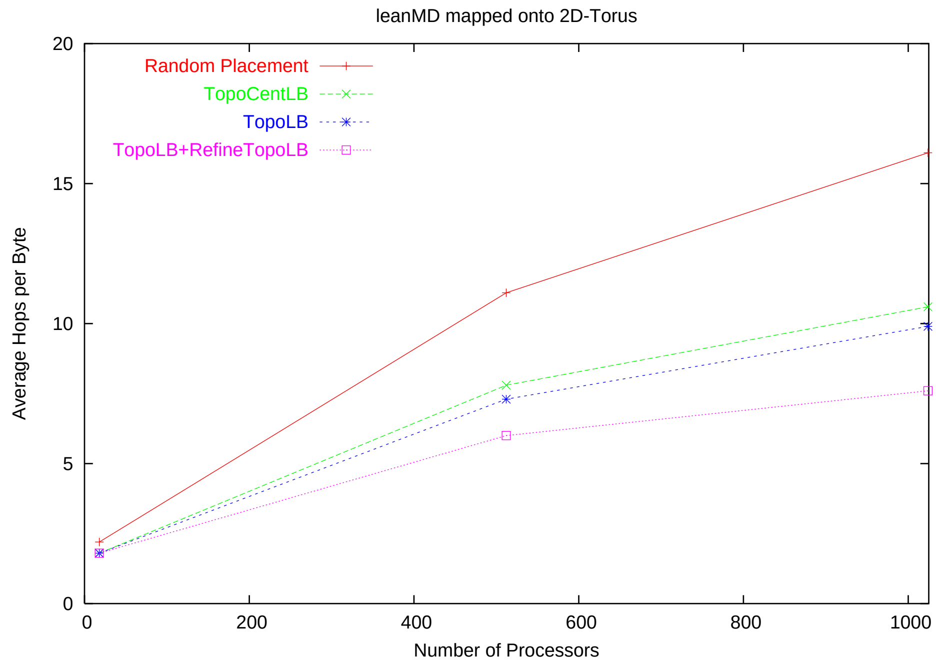
TopoCentLB also results in small values of hops-per-byte which are about 10% higher than those from TopoLB.

4.2.3 LeanMD mapped onto different topologies

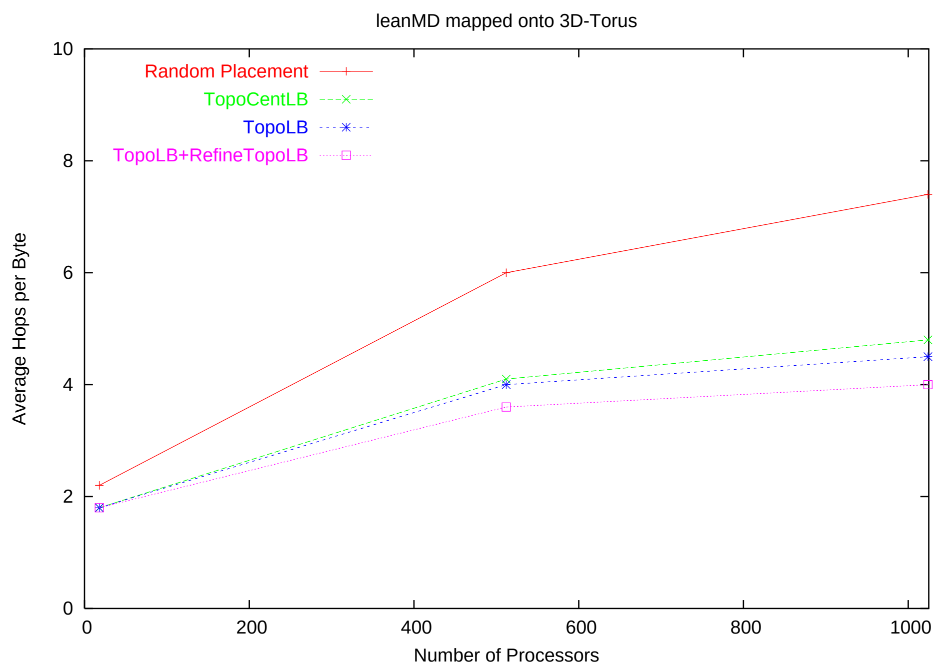
This section will describe the results of mapping communication pattern from a real molecular dynamics simulation program called LeanMD [27]. We have load information dumps for LeanMD on different numbers of processors. The total number of chares is $3240 + p$ where p is the number of processors. This 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 4.4(a) 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. As seen from figure 4.4(a), TopoLB leads to a 34% reduction in average hops-per-byte over random placement. 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.

Figure 4.4(b) 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



(a) Avg. hops-per-byte for LeanMD mapped onto 2D-tori



(b) Avg. hops-per-byte for LeanMD mapped onto 3D-tori

Figure 4.4: Comparison of different mapping strategies on LeanMD data

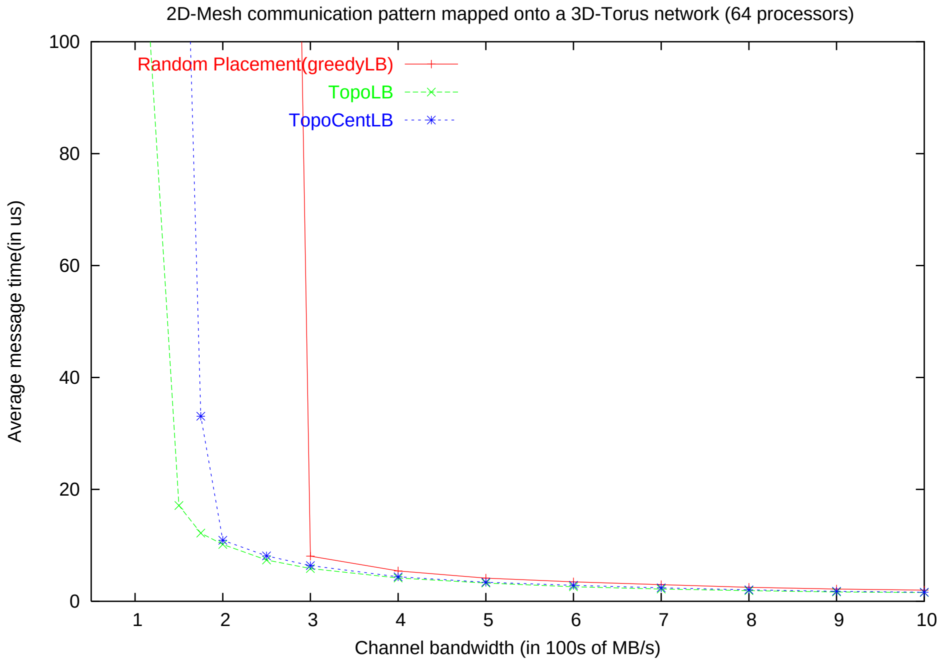
leads to a reduction in hops-per-bytes in the 40% range.

4.3 Performance on interconnection networks

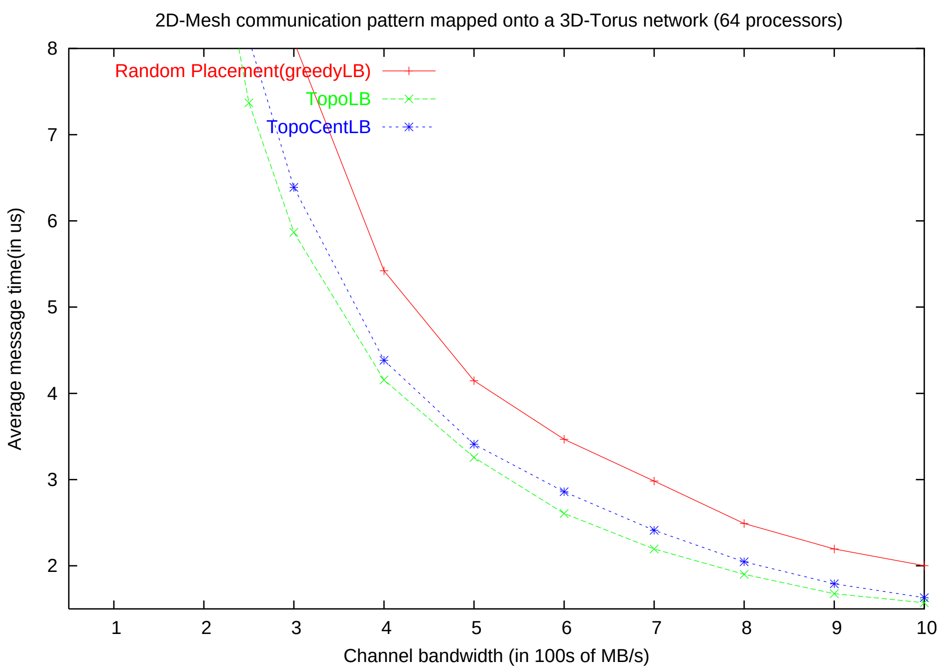
In section 4.2 we discussed the reduction in the average number of hops that each byte travels over the network. In this section we will discuss how this reduction in the hops-per-byte metric translates into gains in execution time and and other characteristics on the network.

We will perform simulations using BigNetSim [36], 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 (essentially random placement) and event traces are obtained. These event traces contain timestamps for message sending and entry point 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.

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 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 4.5(a) 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



(a) Average message latency using different mappings



(b) Detailed comparative view of average latency in the un-congested domain

Figure 4.5: Network simulation results for 2D-mesh communication pattern onto a 64-node 3d-Torus

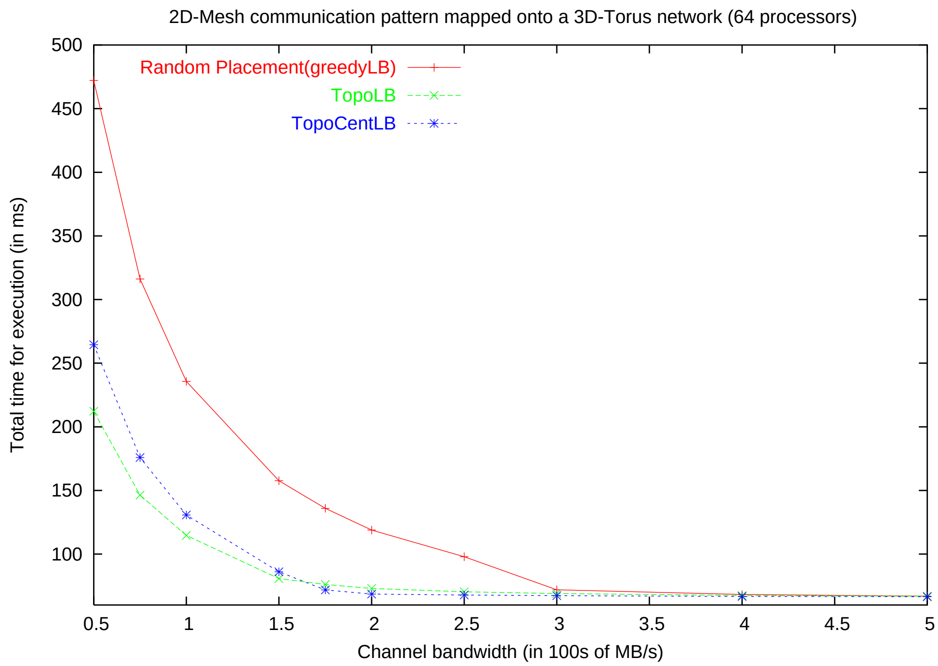


Figure 4.6: Completion time for the execution of 2000 iterations

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 4.5(b) shows the zoomed-in view of figure 4.5(a) 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. This is due to the fewer number of links that a message travels on an average, if TopoLB is used.

The total time for the entire execution to finish is also improved by using intelligent mapping. Figure 4.6 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 in the range 10-25%.

Chapter 5

RefineKLB: A refiner with bounded migration

Dynamic load balancing typically involves periodic calls to the runtime to remap the compute objects. The new remapping can utilize the instrumented load information in the entire or a part of the duration since the last call. A simple approach for remapping is to start each time from scratch without caring for the current location of objects. In this approach, each call to the load balancer is essentially treated as independent and any ab-initio load balancing strategy can be used. However, being oblivious of previous assignment, such an approach would expectedly lead to the migration of a large fraction of the objects. If the load balancer is called with a low frequency, this overhead may be acceptable. However, in a situation where the load scenario changes relatively quickly, frequent load balancing may be required, making the overhead unacceptable. Such frequent change may be due to the underlying nature of parallel application. For example, a simulation of a block of material turning from elastic to plastic starting from an end will see quick load change as compute objects representing regions of the block change state. Frequent calls to load balancer are required, which may lead to large migration overhead. This suggests an approach where the previous mapping is taken into consideration and only some of the objects are moved to achieve load balance. This is called load *rebalancing* through a *refinement* of the current mapping.

5.1 The load rebalancing problem

Informally, the objective of rebalancing is to achieve load balance without causing too much overhead of migration. This overhead could be modeled as the total number of objects that are moved from their previous host processor. Under this model the load rebalancing problem can be stated as: Given a mapping of objects to processors, minimize the maximum load on any processor by moving only K objects away from their processor under the mapping. This is the model used by G. Agarwal, et al [3] and it is shown that the problem is NP-hard. Thus, approximation algorithms are explored and a 1.5-approximation is given [3]. The approximation result can be stated as:

Given a mapping of n movable objects onto p processors and a number k ($\leq n$), the algorithm moves at most k objects and attains a maximum load that is within a factor of 1.5 *of the maximum load if k objects were moved optimally.*

5.1.1 Application to Charm++

The approximation result cannot be directly applied in context of Charm++ load balancing because it assumes that all objects are migratable. However, in Charm++, some objects are not migratable. Also, processors have a background load which is not considered. Background load can also be modeled as an immovable object. In this chapter we present a modification to the algorithm in [3] to account for immovable objects and background load while still ensuring a 1.5-approximation w.r.t. the best possible rebalancing.

A practical problem with the use of a 1.5-approximation result in Charm++ load balancing relates is that that in a large number of load balancing scenarios a factor of 1.5 can be achieved by moving only a few objects. In many scenarios, the initial load imbalance may not even be as large as 1.5 times the average load. With the 1.5-approximation algorithm, we found that even if it is given the freedom to move a large fraction of objects, it ends up moving only just a few objects so as to enable it to guarantee the approximation factor of 1.5. So in a sense the algorithm is *conservative*. With the available residual moves, a heuristic improvement can be undertaken to improve the load balance, albeit without an improvement in the guarantee. In Charm++, the sizes of individual objects are typically small compared to the average load on a processor. In the absence of *large* objects, heuristic improvements usually result in much better load balance than the guarantee of a factor of 1.5.

The Charm++ load balancing framework allows for the specification of an ab-initio balancing strategy and a rebalancing strategy. In a typical usage scenario, the ab-initio strategy is used for the first load balancing cycle , while the later requests are handled by the rebalancing strategy.

5.2 RefineKLB

We implement the modified 1.5-approximation algorithm and a simple heuristic (described in this section) as a Charm++ load balancing strategy called RefineKLB.

5.2.1 Modification of the original algorithm

We will discuss a small modification to the algorithm in [3] to account for background load and un-migratable objects. For each processor, we calculate the total size of un-migratable objects and the background load and replace them by a *single* un-migratable object of this size. Next we run the same algorithm as in [3], although calculating some parameters a little differently. The description of the algorithm below is based on that in [3]. It assumes that the optimal makespan achievable with K moves, OPT, is known. This assumption can be dispensed away later.

Definitions

An object is called *large* if its size is larger than $\frac{1}{2}OPT$, otherwise it is called *small.* We note that there cannot be more than p *large* objects, otherwise a makespan of OPT would not be achievable. A processor is called *large-free* if it does not have any large object. Let L_T be the total number of large objects and p_L be the total number of processors with atleast one *large* object. Let $L_E = L_T - p_L$ denote the number of *excess* large objects. We note that each processor has at most a single *un-migratable* object. This un-migratable object can be large. All other objects are considered to be *migratable.*

Algorithm Mod-PARTITION

1. From each of the p_L processors with large objects, remove all but one large objects as follows:

   - If it contains an un-migratable *large* object, remove all other *large* objects.
   - If it doesn't contains an un-migratable large object, remove all *large* objects except the smallest-size *large* object.

2. For each processor i, caculate the following values w.r.t. to the current configuration:

   - a_i: the minimum number of *migratable small* objects to be removed so that the total size of small objects is at most $\frac{1}{2}OPT$.
   - b_i: the minimum number of *migratable* objects to be removed so that the total size of objects is at most OPT.
   - $c_i = a_i - b_i$

3. Select the L_T processors with the smallest values of c_i (break ties in favor of processors with large objects). Remove a_i small migratable objects from them, so that the total size of small objects is at most $\frac{1}{2}OPT$.

4. From the other $p - L_T$ processors, remove b_i migratable objects so that the total load on each is at most OPT. In doing so, some large migratable objects may be removed from these processors. Assign them to distinct large-free processors created in Step 3.

5. Assign large objects removed in Step 1 to distinct large-free processors created in Step 3.

6. Assign the small objects removed in Steps 3 and 4 one by one to the current least-loaded processor.

Theorem 1: Algorithm Mod-PARTITION achieves an approximation ratio of $\frac{3}{2}$.

Proof: The same proof as in [3] holds for the modified algorithm as well. We just present an outline here. First we observe that the maximum load on any processor after Step 5 is at most $\frac{3}{2}OPT$. After Step 5, there are $p - L_T$ processors with total load at most OPT. The other L_T processors have at most one large object and the total size of small objects at most $\frac{1}{2}OPT$. In Step 6, small jobs are assigned one by one to the current least-loaded processor. Since the total size of all objects is no more than $p \times OPT$, there will be at least one processor with total load not exceeding OPT under any partial assignment. Consequently, assignment of small jobs (which have sizes at most $\frac{1}{2}OPT$) to the current least loaded processor (with load at most OPT) cannot increase the processor's load beyond $\frac{3}{2}OPT$. Thus the maximum load on any processor after Step 6 is at most $\frac{3}{2}OPT$.

Lemma 1 : The number of objects relocated by mod-PARTITION is no more than an optimal algorithm which achieves a makespan of OPT.

Proof: This Lemma is same as Lemma 4 in [3] whose proof holds here for the modified algorithm as well. It is based on the idea of *half-optimal* configurations [3] and we refer the reader there.

Now we will do away the assumption about the knowledge of the value of OPT in advance. First we observe that the behavior of mod-PARTITION changes only when the value of L_T changes, or the values of a_i or b_i change for some processor. These values do not change for small changes in the value of OPT. It is only when OPT changes across some thresholds that the behavior changes. For example an object of size s transforms from being *large* to *small* as OPT increases across $2s$. It is shown in [3] that there are a total of $O(n)$ threshold values which change the behavior. We can try the mod-PARTITION algorithm with OPT value set as these threshold values (in ascending order) until the total objects migrated is atmost k.

5.2.2 Practical strategy for Charm++ load balancing scenarios

We addressed the issue of background load and un-migratable objects in the context of Charm++ in section 5.2.1. However the issue of the algorithm being *conservative* still remains. While a guarantee of load balance within a factor of 1.5 of the optimal is good, we will see that simpler heuristics (with a worse guarantee) performs better in practice. In Charm++, the total number of objects, n, is usually much larger than the number of processors, p. Hence the number of objects on each processor, $\frac{n}{p}$, is large. Thus the size of any individual object is a small fraction of the average load on the processors, making it possible to achieve a makespan that is close to the average load on the processors. Thus a makespan which is 1.5 times the average load is trivial in this scenario.

We have implemented a simple load rebalancing heuristic [3], which guarantees an approximation ratio of 2, but performs better in practice. The following heuristic is applied for the residual moves available after the application of the algorithm in section 5.2.1:

1. Repeat K times:

   Pick the largest object from the current most loaded processor.

2. Place the removed objects one by one, in decreasing order of sizes, on the current least

loaded processor.

It is easy to see that in the case where all objects are the same size, this simple heuristic performs optimally. We will discuss the performance of this heuristic on Charm++ load scenarios in section 5.3.

5.3 Experimental results

This section will describe the performance of RefineKLB load balancing scheme. RefineKLB admits a bound on the number of objects that it is allowed to migrate, as a runtime parameter (using +LBNumMoves K , where $K \in [0, 100]$ is the percentage of objects that can be migrated).

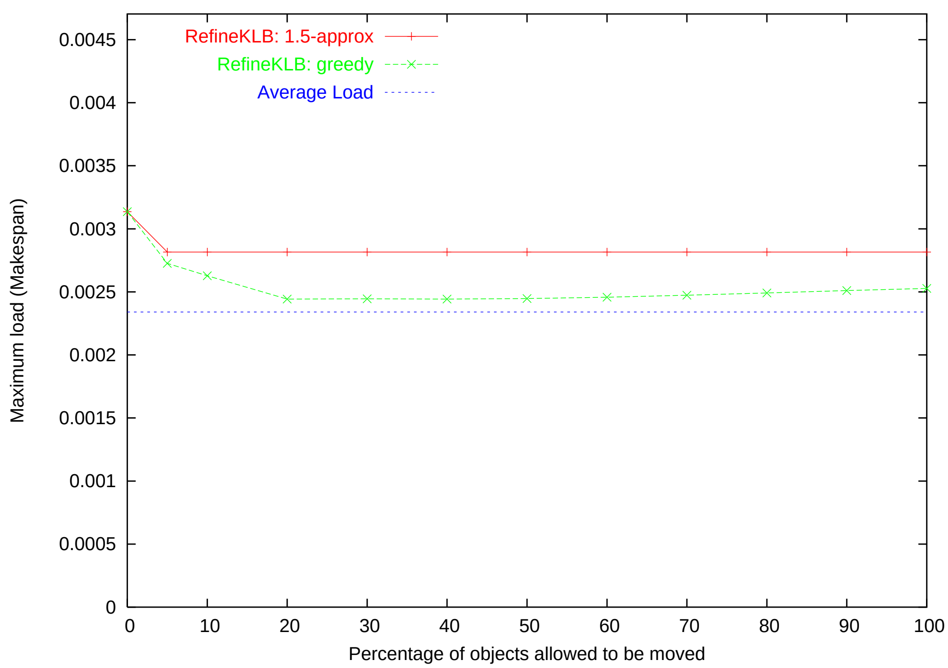


Figure 5.1: Comparison of rebalancing schemes on data from $10k$ objects on $1k$ processors

In this section we will refer to the 1.5-approximation algorithm described in section 5.2.1 as *RefineKLB 1.5 approx*. The other rebalancing scheme, described in section 5.2.2, which performs simple rebalancing in a greedy fashion, will be referred as *RefineKLB greedy*. We will be comparing the two schemes using load database dumps generated from a Charm++ test program called *lb_test* that takes the number of objects to be created as a parameter. For this bechmark program, we use the BigSim [35] simulation environment of Charm++ to obtain dumps for very large number of processors. Performance results for load information gathered from a real molecular dynamics application, LeanMD, will also be considered.

Figure 5.1 shows the performance of the two schemes on data from $10k$ objects on $1k$ processors. It can be seen from the figure that, as a larger percentage of objects are allowed to be migrated, *RefineKLB greedy* utilizes the additional freedom and improves the load balance. On the other hand, *RefineKLB 1.5 approx* doesn't utilize the additional freedom effectively. Although it still achieves an approximation ratio of 1.5, a much better load balance is achievable here.

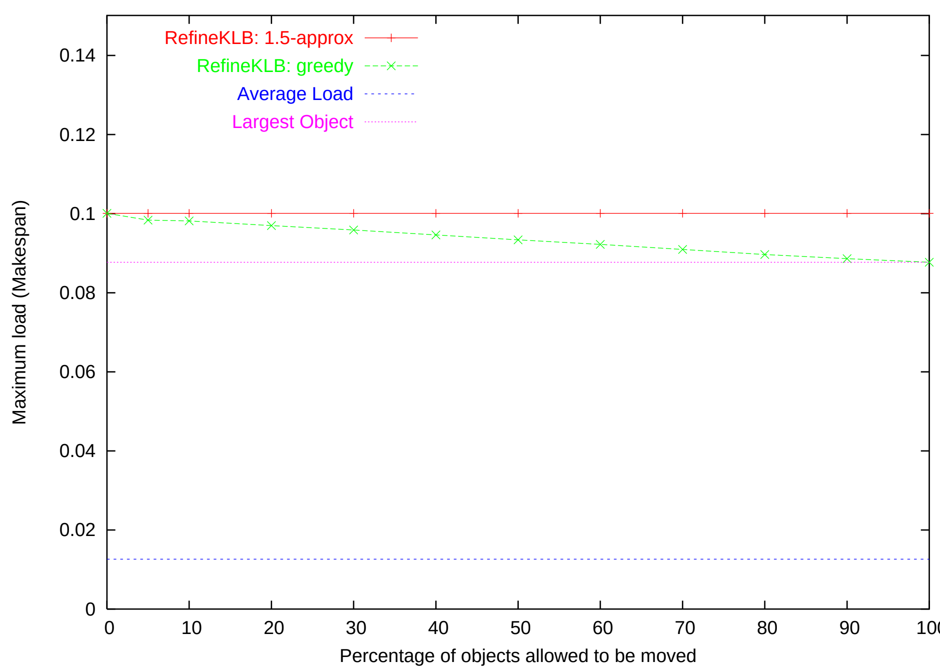


Figure 5.2: Comparison of rebalancing schemes on data from $32k$ processors and $800k$ objects

Similar behavior is seen on data from $32k$ processors. Figure 5.2 compares the two schemes when the number of objects is $800k$. In this particular case, *RefineKLB 1.5 approx* doesn't do anything at all. As shown in the figure, there is one very large object that makes it

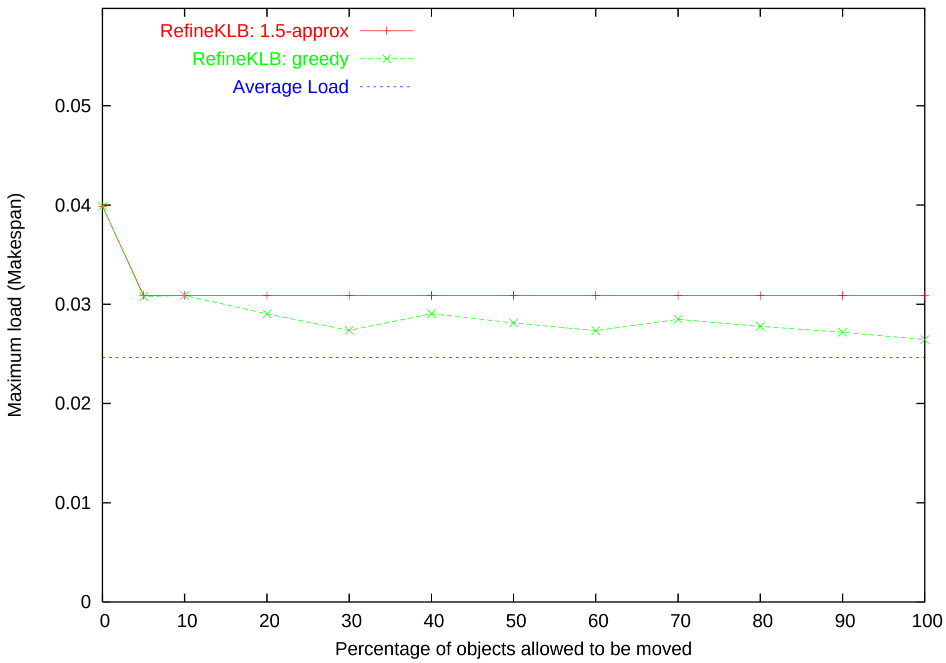


Figure 5.3: Comparison of rebalancing schemes on data from 64k processors and 256k objects

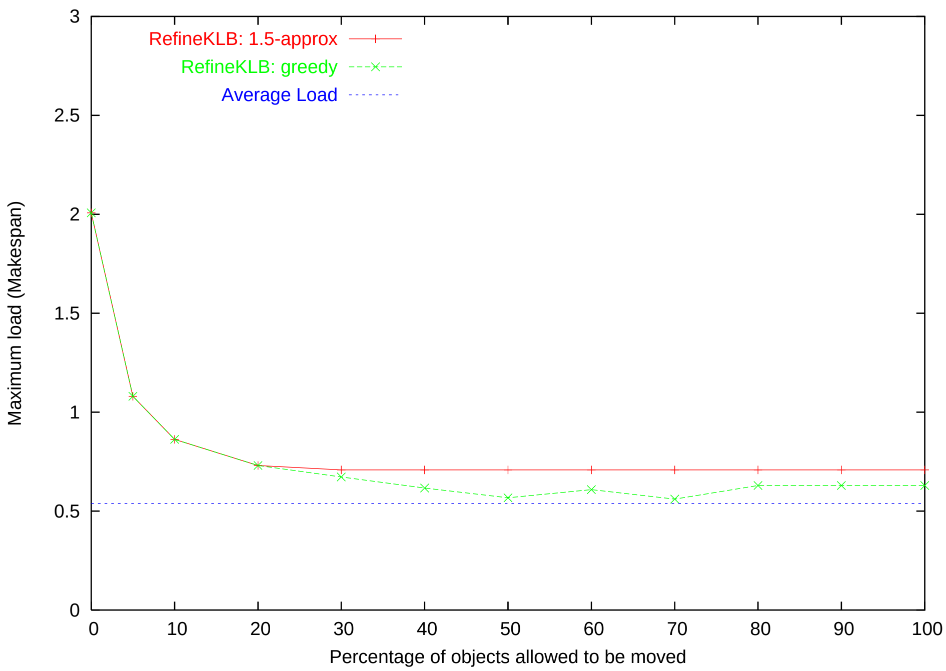


Figure 5.4: LeanMD with 4264 objects on 1k processors.

impossible to achieve a makespan closer to the average processor load. However *RefineKLB greedy* reduces the makespan progressively, to the size of this large object, as more and more migrations are allowed. Figure 5.3 shows the results for $256k$ objects on $64k$ processors. In this case also, greedy refinement beats the 1.5-approximation scheme.

Next, we show the comparison results for the load information obtained from a real application called LeanMD, which is a molecular dynamics program. The performance of the two rebalancing schemes is compared in figure 5.4. The LeanMD data that we are considering is from a run on $1k$ processors, and contains 4264 objects. As seen in figure 5.4, *RefineKLB greedy* performs better than the 1.5-approximation. While the two schemes exhibit similar performance when the percentage of objects that can be migrated is small, the *RefineKLB 1.5 approx* reaches a plateau at 20%. When more than 20% of the objects are allowed to be migrated, *RefineKLB greedy* is seen to lead to a smaller makespan in this case.

Thus, we see that the simple greedy rebalancing heuristic performs better than the 1.5-approximation algorithm under usual scenarios that arise in rebalancing object computation load in Charm++ applications.

Chapter 6

Summary and future work

This thesis presents load balancing schemes aimed at solving two categories of problems that arise in the context of parallel programming.

A task-mapping algorithm, called TopoLB, is described in this thesis. The algorithm provides a solution to the problem of mapping tasks onto physical processors connected in a given topology, so that most of the communication is 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 optimally in many cases, although it does not consider the shapes of the graphs specifically. We show, via simulations, that an efficient mapping that reduces the total communication load on the network, or hop-bytes, leads to lower network latencies on an average, and provides better tolerance of network bandwidth constraints.

In the future, gains from topology-aware task mapping should be studied on real large parallel machines, like BlueGene (BG/L). Due to massively large sizes of these machines, a distributed approach toward keeping communication localized in a neighborhood may be needed for scalability. A hybrid approach, such as that in [34], could also be investigated.

We also study the problem of rebalancing load by migrating a bounded number of objects from their current host processors. Towards this end, a modification is proposed to the algorithm in [3], to account for the background load on processors and the possibility of un-migratable objects. We find that in typical scenarios in load balancing Charm++ applications, or situations where objects sizes are small in comparison with average processor load, a simple greedy heuristic performs better. Although the greedy heuristic provides a weaker guarantee (2-approximation) than the other algorithm (1.5-approximation), it leads to better rebalancing in practice.

A better model for rebalancing cost needs to be considered in the future. Since the actual migration of objects takes place in a distributed fashion in parallel, a more accurate cost metric would be the maximum number of migrations that any individual processor is involved in (as either a source or a destination). Strategies need to be designed to reduce this migration cost metric.

References

[1] *30th International Workshops on Parallel Processing (ICPP 2001 Workshops), 3-7 September 2001, Valencia, Spain*. IEEE Computer Society, 2001.

[2] 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.

[3] Gagan Aggarwal, Rajeev Motwani, and An Zhu. The load rebalancing problem. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–265, New York, NY, USA, 2003. ACM Press.

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

[5] Takanobu Baba, Yoshifumi Iwamoto, and Tsutomu 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.

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

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

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

[9] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Blni, Albert I. Reuther, Mitchell D. Theys, Bin Yao, Richard F. Freund, Muthucumaru Maheswaran, James P. Robertson, and Debra Hensgen. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *HCW '99: Proceedings of the Eighth Heterogeneous Computing Workshop*, page 15, Washington, DC, USA, 1999. IEEE Computer Society.

[10] 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.

[11] 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.

[12] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[13] Ronald P. Bianchini Jr. and John Paul Shen. Interprocessor traffic scheduling algorithm for multiple-processor networks. *IEEE Trans. Computers*, 36(4):396–409, 1987.

[14] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[15] L. V. Kalé. Comparing the performance of two dynamic load distribution methods. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, St. Charles, IL, August 1988.

[16] L. V. Kale, Milind Bhandarkar, and Robert Brunner. Run-time Support for Adaptive Load Balancing. In J. Rolim, editor, *Lecture Notes in Computer Science, Proceedings of 4th Workshop on Runtime Systems for Parallel Programming (RTSPP) Cancun - Mexico*, volume 1800, pages 1152–1159, March 2000.

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

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

[19] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

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

[21] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint 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.

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

[23] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graph. *Bell System Tech. Journal*, 49:291–307, Feb. 1970.

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

[25] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Math. Program.*, 46(3):259–271, 1990.

[26] N. Mansour, R. Ponnusamy, A. Choudhary, and G. C. Fox. Graph contraction for physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 1–10, New York, NY, USA, 1993. ACM Press.

[27] Vikas 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.

[28] Juan M. Orduña, Federico Silla, and José Duato. A new task mapping technique for communication-aware scheduling strategies. In *ICPP Workshops* [1], pages 349–354.

[29] Sayantan Chakravorty, Celso Mendes and L. V. Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA 2005*, 2005.

[30] W. W. Shu and L. V. Kalé. A dynamic load balancing strategy for the Chare Kernel system. In *Proceedings of Supercomputing '89*, pages 389–398, November 1989.

[31] Bernd Stramm and Francine Berman. Communication-sensitive heuristics and algorithms for mapping compilers. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 222–243, New York, NY, USA, 1988. ACM Press.

[32] Kenjiro Taura and Andrew 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.

[33] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, September 1993.

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

[35] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[36] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, number to appear, 2005.