RC24290 (W0609-052) September 15, 2006
Computer Science

IBM Research Report

Optimizating Communication in MPI Programs for an Execution Environment

Peter F. Sweeney, Robert W. Wisniewski, Calin Cascaval, Stephen E. Smith
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Optimizing Communication in MPI Programs for an Execution Environment

Peter F. Sweeney Robert W. Wisniewski Călin Caşcaval Stephen E. Smith
IBM T.J. Watson Research Center

Abstract

Message Passing Interface (MPI) is the commonly used programming paradigm for high performance computing (HPC). The model has become popular mainly due to its portability and support across HPC platforms. Because MPI programs are written in a portable manner, programmers optimize application-related aspects, such as the algorithm and generic communication, but typically do not optimize for the execution environment. In particular, MPI tasks are often mapped to the processors in a linear order.

In this paper, we show that mapping tasks to processors in an MPI program is a critical decision that significantly impacts performance. We present techniques to model the hardware communication topology and application communication patterns. Given such a model, we describe an algorithm to estimate the communication cost of any mapping of MPI tasks to processors. Also based on the communication model, we present a heuristic algorithm to generate a mapping of MPI tasks to processors. We demonstrate that these generated mappings improves overall performance by up to 35%.

1. Introduction

Message Passing Interface (MPI) is the prevalent programming model for high performance computing (HPC), mainly due to its portability and support across HPC platforms. Because most HPC centers have a large variety of machines, portability is a major concern of MPI programmers. Therefore, MPI programs are typically optimized for algorithmic and generic communication issues. Parallel computation models such as the LogP [9] and LogGP [4], allow users to analyze parallel algorithms by providing a small set of parameters that characterize an abstract machine. Often times in such models, the execution environment (machine-specific) characteristics are ignored by design. For example, the LogP model intentionally leaves out the intercommunication network characteristics and the network routing algorithm in order to keep the model tractable.

However, in many cases, for a particular application running on a specific machine, the mapping of MPI tasks to processors can have a significant impact performance. This effect is due in part to the fact that many scientific applications exhibit a regular point-to-point communication pattern between a subset of the neighbors[1]. The default, linear order of mapping MPI tasks to processors, which is often used in practice, does not consider the application's regular communication pattern when mapping MPI tasks to processors and therefore may not achieve the best performance, as we will show later.

In this paper, we focus on finding a good mapping of MPI tasks to processors. To address this problem, we need to understand and model the hardware communication topology of the execution environment and the application communication pattern. We define the *hardware communication topology* (HCT) as the hardware components used by the executing MPI tasks to communicate. For example, MPI tasks on cores in a chip communicate through caches, cores in different chips communicate through memory; and cores in different nodes communicate through a network interconnect. The HCT encompasses all of these communication links. The *application communication pattern* (ACP) characterizes how MPI tasks exchange data with one another. This includes the number of messages and their size. (Section 3 provides more details).

Our goal is to define a small set of parameters that characterize both the hardware communication topology and the application communication pattern. We demonstrate that a model developed with these parameters can estimate the communication cost of different mappings of MPI tasks to processors for a given HCT and ACP. This cost estimator can be embedded in heuristic algorithms to guide their mapping choices or can be used to compare different mappings to determine which one is more advantageous. In this paper, we present a model, a heuristic and a cost estimator, and evaluate their effectiveness in improving performance on a set of benchmarks.

This paper makes following contributions:

- defines a set of parameters that characterize a hardware communication topology and an application communication pattern (Section 3), and presents an algorithm that uses these parameters to derive the communication cost for a particular mapping of tasks to processors (Section 5);
- presents an algorithm for mapping tasks to processors based on a greedy heuristic (Section 4); and
- demonstrates that mapping tasks to processors has a significant effect on performance, and show that up to 35% overall performance improvement can be achieved automatically using our approach (Section 6 and Section 7).

[1] This is partly a consequence of good MPI programming education – if global communication is needed, MPI programmers use collective operations over defined MPI communicators, which are typically tuned to the underlying architecture.

We discuss related work in Section 8, future work in Section 9, and conclude in Section 10.

2. Motivation

A large class of scientific applications are written in a stylized manner. Typically these applications consist of a series of steps, where in each step, the application executes two phases. A computation phase in general is followed by a communication phase with synchronization between them. This class of applications has *concurrent communication* where the communication between MPI tasks occurs at the same time, i.e., there is a period of time during which all of the tasks are performing communication.

In this class of applications, there are four places where performance can be improved, defined as follows:

computation the amount of time required to perform the computation.

computation imbalance the difference between the longest and shortest compute times per phase.

communication overhead the amount of time required to perform communication.

communication imbalance the difference between the longest and shortest communication times per phase.

In this paper, we focus on reducing the communication overhead and communication imbalance in applications using MPI. The goal is to improve performance by adapting an application to its execution environment, without modifying the source code. We achieve this goal by mapping MPI tasks to processors to exploit the hardware communication topology with respect to bandwidth and concurrency. Grouping together tasks that are communicating frequently reduces the communication overhead of point-to-point communication by exploiting the higher bandwidth available in the lower levels of the hardware communication topology (HCT). We exploit concurrency in the network by separating groups of tasks such that intergroup communication can proceed in parallel.

To automate the mapping process, we need to be able to estimate the performance of different mappings for a particular application running on a given HCT. The data in Figure 1 illustrates the mapping of tasks to processor significantly affects the communication bandwidth and in turn the performance of the application. In particular, we have abstracted out four characteristics that are crucial to modeling communication cost. They include message length, HCT concurrency, maximum bandwidth, and placement of tasks.

The graph in Figure 1 illustrates the bandwidth (y-axis) for different message lengths (x-axis) when pairs of MPI tasks communicate concurrently. We obtained this data using NetPIPE [17] (http://www.scl.ameslab.gov/netpipe/), a network protocol independent evaluator and mapping the MPI tasks to different cores on a two-tiered network of eight PowerPC 970FX dual-processor Xserve machines. The graph plots the following four communication configurations: the lowest line represents the unidirectional bandwidth when two tasks are mapped to processors on different machines in different subnets (labeled “2T across subnets”); the next higher line represents the unidirectional bandwidth when two tasks are mapped to processors on different machines in the same subnet (labeled “2T in subnet”); the next line represents the bidirectional bandwidth when two tasks are mapped to processors on different machines in the same subnet (labeled “2T B in subnet”); and the highest line represents when four tasks are mapped to processors on different machines in the same subnet (labeled “4T B in subnet”).

The graph illustrates that bandwidth varies with message length. For example, as message length increases up to 64 KB, bandwidth increases for all four configurations, then drops significantly as the MPI implementation switches message delivery mechanisms. Because bandwidth varies across message sizes, a model for communication cost should take message length into consideration.

The top two lines, “2T B in subnet” and “4T B in subnet”, show that when the number of communicating tasks doubles from two to four and each pair of communication tasks does not share resources in the switch, the bidirectional bandwidth for a message length also doubles. The doubling of bandwidth is due to parallelism provided by the switch [2]. It is a common characteristic of switches to have a higher internal bandwidth than on each of the ports. Because the internal bandwidth of a switch allows multiple concurrent ports to communicate, a model for communication cost should take HCT concurrency into consideration.

HCT resources, switches or memory, have a maximum bandwidth between any given pair of communicating tasks. Figure 1 illustrates multiple instances of communication throughput being bound by maximum bandwidth. As message lengths exceed 64 Kb, the bandwidth is bound by 100 Mbps for all configurations other than “4T B in subnet”, which is bound by 200 Mbps [3]. Because bandwidth is bound, a model for communication cost should take maximum bandwidth of the HCT resources into consideration.

The graph illustrates that the mapping of tasks to processors impacts bandwidth. For example when the message length is less than 64Kb, mapping two tasks to processors in the same subnet has improved bandwidth over mapping the same two tasks to processors in different subnets. Therefore, a model should take the mapping of tasks to processors into consideration.

One other component, which is needed for estimating the performance of a mapping, is the application communication pattern. This will be discussed in detail in the next section.

Finally, the problem of finding the best mapping for a given hardware communication topology and application communication pattern is NP complete [18]. Any MPI task may be mapped to any processor. Thus, for T tasks there are $T!$ mappings. Because there is an exponential number of mappings, any reasonable-sized problem requires a heuristic algorithm to determine a mapping. We present such a heuristic algorithm in Section 4.

3. Data Model

This section presents the model of the hardware communication topology (Section 3.1) that MPI tasks use to communicate, and the model of application communication patterns (Section 3.2), the data that these tasks communicate.

3.1 Hardware Communication Topology

Today’s large-scale machines are often constructed out of smaller SMP nodes connected in a hierarchical manner by a high bandwidth interconnect. This section presents how we model such a tree-structured hardware communication topology.

Figure 2 illustrates the two basic elements that we use to model a hardware communication topology. A processing element represents a single computational unit (denoted as CU in the figure) at the lowest level in the hardware communication topology. For single threaded machines, a computational unit is a processor (or core), for SMT machines it is the hardware thread. A switch ele-

[2] Specifically, we used an NETGEAR FS108 network switch that has 8 switched 100 Mbps ports.

[3] We use the LAM MPI implementation (www.lam-mpi.org) that uses a hand-shaking protocol for messages whose length equal or exceed 64 kb. The hand-shake protocol is a bidirectional, rendezvous protocol that requires the two tasks, prior to communication ensure that the receiving task has a buffer available to receive the message. The hand-shake protocol effectively converts bidirectional communication to unidirectional due to the protocols interference.

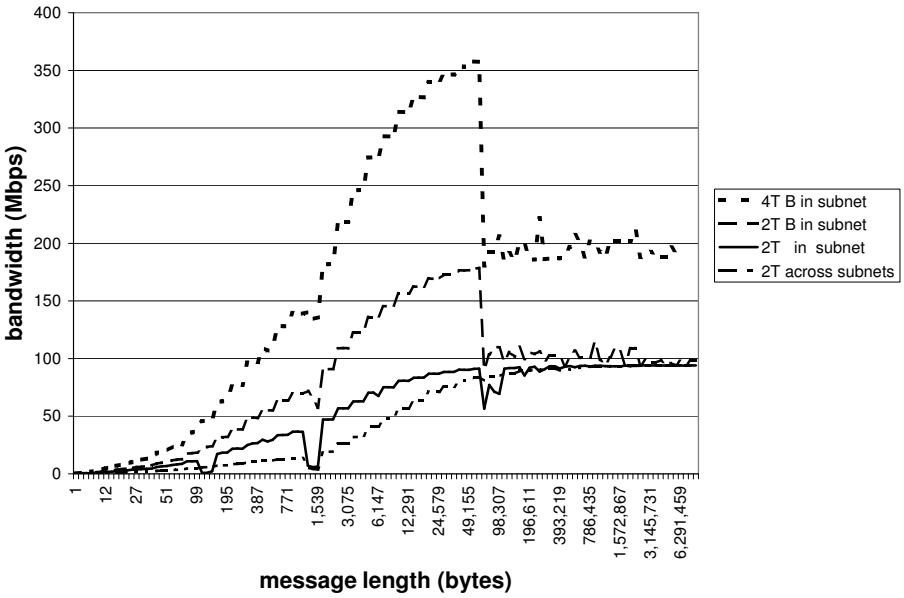


Figure 1. Throughput graph: Message Length versus Bandwidth.

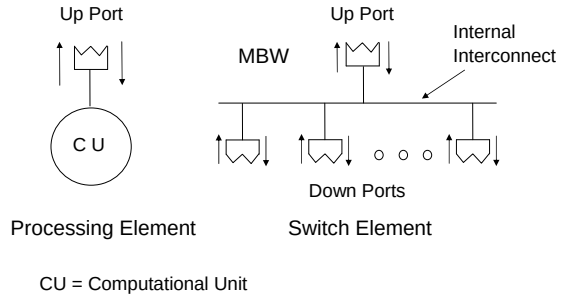


Figure 2. Model Elements

ment is used to combine lower-level components, either processing or switch elements. The switch's "down ports" represent how lower-level components that are connected to the switch communicate. The single "up port" represents how a component communicates with a higher-level switch. The hierarchical interconnect forms a tree-structured topology.

The arrows associated with a port represent directional bandwidth as a function of message length and the maximum bandwidth that port can support in a particular direction. The down arrow represents "in" bandwidth and the up arrow represents "out" bandwidth. This is used to model the fact that bidirectional communication through a port can occur in parallel. In addition, each pair of communicating tasks has the full bandwidth of a port they are using provided no other tasks are using the same port. If a port is shared between pairs of communicating processing elements, then the bandwidth of the shared port is bounded by the port's maximum bandwidth.

A switch element has a maximum bandwidth (identified as MBW in the figure) that bounds the aggregate bandwidth of all the ports on the switch. Maximum bandwidth determines the switch element's concurrency. If no arrow is associated with a port, the switch element's maximum bandwidth bounds the port's bandwidth.

A hardware communication topology may define multiple processing elements to reflect multiple types of computation units and define multiple switch elements to reflect multiple types of switch elements. For example, the hardware communication topology that we used in our experiments has two switch elements defined, one for a PowerPC 970FX dual-processor Xserve machine, and another for a NETGEAR FS108 switch.

3.2 Application Communication Pattern

An *application communication pattern* characterizes the way in which one MPI task exchanges data with another MPI task. We characterize an application communication pattern by the number of messages of a certain size that is communicated between each pair of MPI tasks. For simplicity, this characterization ignores the order that messages are sent between two MPI tasks and ignores the blocking/nonblocking semantics of the point-to-point communication. In Section 7.3, we discuss how this simplification affects our results.

Application communication patterns are derived from a trace of the point-to-point communication in an MPI application. As we discussed before, typically communication and computation alternate in phases. Such phases repeat throughout the execution of the application. Thus, only one instance of each phase needs to be modeled. To obtain the best mapping, each unique class of phases would need to be characterized with the weighted combination of those phases being used to determine the overall effect on performance.

4. Mapping Algorithm

Input:
  T: hardware communication topology
  P: application communication pattern between MPI tasks
Output:
  M: mapping of MPI tasks to processors in T
Method:
  // Initialization: assign a MPI task to a cluster
  for $i = 1$ to $P.numberOfTasks$
    C_0 += $\{< i, 0 >\}$

  // exploit bandwidth
  for i = 1 to $T.levels$
    C_i = {} // cluster set for level i
    S = {} // communication set
    forall Q: T_i.numberOfChildren combinations of C_{i-1}
      $S+ = \{< Q, b >: b =$ number of bytes communicated
            between MPI tasks in different $Ci - 1$ clusters $\in Q\}$
    foreach $< Q, b > \in S$ in sorted order by b
      $C_i + =< Q, b >$ if no MPI task in Q already $\in C_i$

  // exploit concurrency
  for i = $T.levels$ to 1
    for j = 1 to $T_i.numberOfChildren$
      $T_i^j.cluster = C_{i-1}^j.first$

  // output
  forall $n \in T_0$
    $M+ =< n.cluster, n.processor >$
  return M

Figure 3. Greedy algorithm to map MPI tasks to processors.

Given a hardware communication topology and an application communication pattern, the mapping algorithm in Figure 3 computes an assignment of MPI tasks to processors. The goal of this mapping is to optimize performance by exploiting bandwidth differences on communication links in the HCT and by exploiting concurrency in different subtrees in the HCT hierarchy. The algorithm uses a greedy heuristic under the following assumptions: i) a single MPI task is assigned to a processor, and ii) that bandwidth stays the same or gets worse as one move higher in the HCT (further away from the processing elements). For presentation purposes, the algorithm presented in Figure 3 makes the following simplifications compared to the one we actually implemented: the arity of the children of a node in the HCT is the same at each level, and the assignment of communication sets that span clusters is ignored.

The algorithm will cluster MPI tasks according to the above heuristics. During initialization, each MPI task is assigned to a separate cluster, C_0. Then, the algorithm makes two passes over the HCT, first to exploit bandwidth, and second to exploit concurrency. The first pass starts from the next to the bottom level and assigns a number of C_{i-1} clusters to C_i clusters (one level up) according to the number of children of an element at the i^{th} level in the HCT. The number of bytes communicated between the tasks in the C_{i-1} clusters determines how C_i clusters are generated: the C_{i-1} clusters with the most inter-cluster communication are created as C_i clusters first. We use clusters in this pass instead of assigning tasks directly to HCT elements to allow the second pass, which assigns tasks to HCT elements, to exploit concurrency. The second pass starts from the top assigning clusters to elements at each HCT level. At the lowest level of the HCT, MPI tasks are assigned to processing elements. The output of the algorithm is a mapping of MPI tasks to processors.

Algorithm Analysis The complexity of this algorithm is bounded by a sort operation on the number of communication pairs of MPI tasks. The bound is $O(R\ log\ R)$, where R is the pairs of MPI tasks that communicate which in turn is bounded by $T * T$ where T is the number of MPI tasks.

Discussion We chose a mapping algorithm that does not use the cost estimator presented in Section 5 to guide the mapping. This decision allows us to validate our cost and the mapping algorithms independently. Another class of heuristic algorithms could use the cost algorithm to guide the mapping. Orthogonally, a probablistic approach could be used to help avoid local minima that greedy algorithms are susceptible to finding. The exploration of the better heuristic algorithm is future work.

5. Cost Estimator

Given a hardware communication topology, a mapping of MPI tasks to processors, and the application communication pattern between these tasks, the cost estimator computes a communication time (or cost) for a communication phase using the algorithm in Figure 4. To simplify the presentation, the cost estimator presented in Figure 4 does not consider the maximum bandwidth of the switch element.

Input:
  T: hardware communication topology
  M: mapping of MPI tasks to processors in T
  P: application communication pattern between tasks in M
Output:
  C: estimated communication cost
Method:
  foreach task src $\in M$
    // compute serial cost for src's messages
    foreach message msg $\in P$ sent by src to dst
      foreach port p $\in T$ on the path from src to dst in T
        // update local copy of p's cost and bytes
        p.taskCost += (msg.len / p.bandwidth(msg.len) * msg.count
        p.taskBytes += msg.len

    // compute aggregate cost at each port
    foreach port p $\in T$
      // update p's cost and bytes with local copy
      p.bytes += p.taskBytes
      bytesAvailable = (p.maxBandwidth * p.cost) - p.bytes
      if bytesAvailable $<$ p.taskBytes
        p.cost += ((p.taskBytes - bytesAvailable) / p.taskBytes) * p.taskCost)

      // clear local copy of p's cost and bytes
      p.taskCost = p.taskBytes = 0

  return $C = MAX$(p.cost) $\forall$ port p $\in T$

Figure 4. Cost estimator computes the estimated communication time.

Before discussing the algorithm in detail, there are several important points to highlight. First, we strove to design a model that was simple, but accurate enough to to predict the performance of a mapping. For application communication patterns, only the number and the size of the messages (`msg.count` and `msg.len` in Figure 4) between two tasks is maintained. The order in which messages are sent is not taken into consideration. For the hardware communication topology, the only parameters in our model are: observed bandwidth per message length, concurrency, and the maximum bandwidth of a port. The values used for these parameters were obtained experimentally using NetPIPE [17] running between different combinations of tasks and processors, as explained in Section 2.

Second, our algorithm takes advantage of concurrent communication. This allows us to remove the ordering of messages when we capture the application communication patterns. The drawback is that, for applications that overlap computation with communication, the cost estimator algorithm will produce less accurate results.

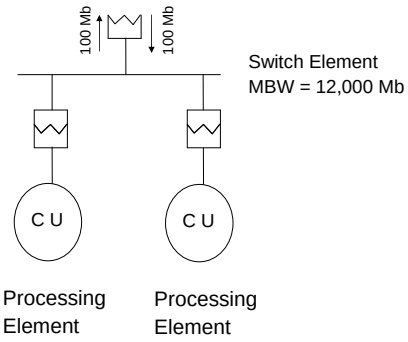


Figure 5. Model of an Xserve: a dual core, single threaded machine. **TODO: Steve remove 100Mb and up and down arrows from CU port. Where is message length information?**

Our cost estimator algorithm is presented in Figure 4. For each task, the cost estimator has two steps. The first step accumulates, for each message, the cost (`p.taskCost`) and bytes (`p.taskBytes`) at each port (`p`)in the hardware communication topology through which the message flows. Because the messages are processed serially and the cost and bytes at each port are accumulated, the maximum bandwidth at any port is guaranteed not to be exceeded for the current task's communication. The second step aggregates the current task's cost (`taskCost`) and bytes (`taskBytes`) at each port, `p`, limiting the aggregation by `p`'s max bandwidth. This step exploits the fact that task communication in different portion of the tree-topology can occur at the same time. `p.bytesAvailable` is the number of additional bytes that can be communicated at `p` without exceeding `p`'s max bandwidth. If `p.bytesAvailable` is less than the number of bytes sent by the messages of the current task at `p` (`p.taskBytes`), additional cost has to be added to `p.cost`. The estimator computes the additional cost as a faction of the task's cost (`taskCost`), where the fraction is computed as `((p.taskBytes - bytesAvailable) / p.taskBytes)`.

After all communication of all tasks is processed, the estimated commuincation cost (`C`) for the communication phase is taken as the maximum time over all ports in the hardware communication topology.

Algorithm Analysis The complexity of this algorithm is $O(M \times N)$ where M is the number of messages sent between MPI ranks and N is the number of ports in the hardware communication topology.

6. Experimental Methodology

In this section, we discuss the experimental framework used to obtain our results. We ran on a two-tiered network of eight PowerPC 970FX dual-processor Xserve machines. Thus the total number of processors available in our experiments is sixteen. The processors are clocked at 2 GHz. Dual-processors share the memory bus but have separate caches. We ran a Gentoo Linux distribution with the 2.6.12-rc2 kernel.

Figure 5 illustrates how we model an Xserve machine using one switch element and two processing elements from Figure 2 in Section 3. Each machine contains two single-threaded cores on a chip, the cores communicate with each other through memory; they do not share caches. The switch element models the memory bus to which the two cores are connected. The up-port of a switch represents the network interface card. The down-ports model how

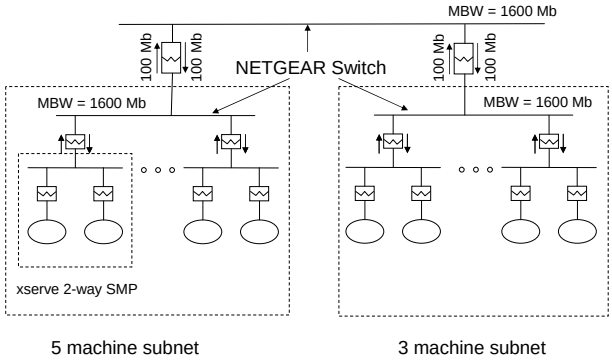


Figure 6. Experimental Hardware Communication Topology

the cores communicate. The bandwidths values were determined by running two NetPIPE processes on different processors in one machine.

Figure 6 illustrates schematically the hardware communication topology used in our experiments. Our experimental network consists of eight Xserve machines, five on one subnet, and three on another subnet. The switches on the subnet were NETGEAR switches and the subnets were connected together via a large Cisco switch with the same properties as the NETGEAR switch, i.e., 100Mbps per port with higher internal connectivity.

Other hardware topologies can be modeled in a similar tree-structured manner. For example, we are in the process of modeling a PowerPC POWER5 machine where each core has two hardware threads, each chip has two cores, and each module has eight chips to provide 32 logical processors that communicate through the memory hierarchy.

6.1 Benchmarks

For our experimental evaluation, we use the MG benchmark as a representative benchmark from the NAS Parallel Benchmarks suite [5, 6] and two large scientific applications UMT2K [19] and RF-CTH [13, 2]. To collect application communication patterns we used the PMPI interface to intercept all MPI calls in the application. For each benchmark, we evaluated different configurations that differed by the number of MPI tasks. For reported execution times, we ran each benchmark five times and took the median.

MG uses a multigrid method to compute the solution of the 3D scalar Poisson equation. The partitioning is done by recursively halving the grid until all the processors are assigned. This benchmark requires a power-of-2 number of processors. Communication occurs between iterations by exchanging data at the borders.

UMT2K is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. It solves the first-order form of the steady-state Bolzmann transport equation. The equation's energy dependence is modeled using multiple photon energy groups, each using a collocation of discrete directions. The memory access pattern varies substantially for each direction, and the entire mesh is "swept" multiple times. The unstructured meshes are generated at run-time using a 2D unstructured mesh and extruding it into the

third dimension. The meshes are distributed across the MPI tasks using the METIS Library [1].

RF-CTH is a code used to explore the effects of strong shock waves on materials using different models. The code simulates shock hydrodynamics equations. The MPI parallel version is based on an SPMD programming model. It decomposes a domain into subdomains and communication between those subdomains uses ghost cells.

7. Evaluation

This section evaluates both the cost estimator presented in Section 5 and the mapping algorithm presented in Section 4. Figure 7 evaluates the accuracy and predictability of the cost estimator. Figure 9 shows that our mapping algorithm has significant performance gains in many cases (up to 35%) with an overall average speedup of ten percent. This is achieved automatically without programmer intervention.

7.1 Cost Estimator Evaluation

We start by evaluating the predictability and accuracy of the cost estimator. *Predictability* determines the cost estimator's ability to determine if a different mapping will speed up or slow down the benchmark's execution. *Accuracy* determines how close is the difference between two mappings predicted by the cost estimator compared to the difference between their actual executions.

The bar graphs in Figure 7 illustrate the cost estimator's predictability and accuracy for the three benchmarks by comparing the estimated execution time speedup (left bar) and actual execution time speedup (right bar). The cost estimator is predictable if both bars go in the same direction, and the algorithm is accurate if both bars are close in height. In determining whether a particular mapping should be chosen, predictability is the important metric. For example, as long as a meaningful performance improvement is predicted, whether it is actually 10% or 15% is less critical that knowing there is a performance win.

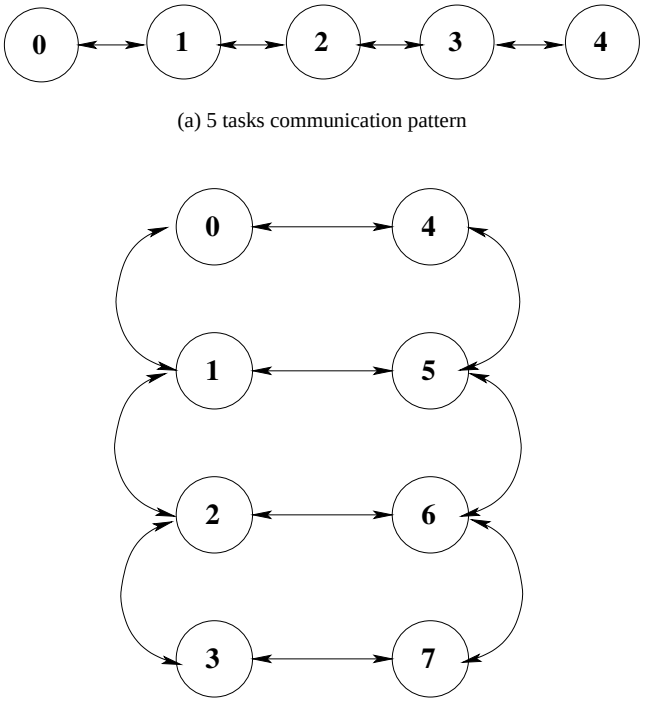
In what follows, speedup reflects the total execution time decrease, not only communication time. We compute the estimated speedup as the percentage difference between the cost estimator's prediction of the default (linear mapping of tasks to processors) and greedy mappings (Section 4) multiplied by the percentage of total execution time spent in MPI communication[4]. We compute the actual speedup as the percentage difference between the actual execution times of the default and greedy mappings. Each configuration, which is determined by a benchmark, number of tasks and a mapping, was run five times and we used the median of the five runs as the execution time. We found that the variation of execution times between runs of a configuration is insignificant.

The top bar graph in Figure 7 evaluates the cost estimator for UMT2K. The graph illustrates that the cost estimator has good predictability. All configurations, except 2 and 4, are predictable. For the two unpredictable configurations the difference between estimated and actual speedups is less than 1%, which is within measurement error. This data illustrates the model achieves the predictability needed to determine which mapping to use.

The graph also illustrates that the cost estimator's accuracy has room for improvement. In some configurations, there is significant variation between estimated and actual speedups. For example, the 9 task configuration has an estimated speedup of 1.5%, but an actual speedup of 12.1%; whereas the 15 task configuration has an estimated speedup of 10.6%, but an actual speedup of 2.2%.

The 16 task configuration illustrates a case where our mapping heuristic does poorly because the heuristic is greedy: the greedy mapping results in more bytes being communicated in the top-level switch in the hardware communication topology than the default mapping. This level has the lowest bandwidth and will take the most time, as is correctly predicted by the cost estimator. Thus, using the cost estimator prediction, we will choose the default mapping as the best mapping.

[4] The total execution time spent in MPI communication includes both collective and point-to-point communication.



(a) 5 tasks communication pattern

(b) 8 tasks communication pattern

Figure 8. RFCTH communication patterns.

The middle graph in Figure 7 evaluates the cost estimator for RFCTH. The graph illustrates that the cost estimator is predictable. In only one configuration out of 15 (8 tasks), does the cost estimator's prediction fails — the estimated speedup is 4.0%, while the actual speedup is -5.9%, a slowdown. We studied the application communication pattern (ACP) for the 8 task configuration and determined that the tasks are laid out two dimensionally as a ladder, as shown in Figure 8(b). There is 75% more communication occurring along the steps than along the sides. The greedy algorithm appears to have a better mapping by clustering the steps on the same machine, where as the default mapping clusters the steps across machines. More work is required to understand why the greedy mapping performs worse than the default mapping for this application communication pattern.

The graph also illustrates that the cost estimator's accuracy continues to have room for improvement: in some configureations, there is significant variation between estimated and actual speedups. For example, the 10 task configuration has an estimated speedup of 9.1%, but has an actual speedup of 29.6%; and the 12 task configuration has an estimated speedup of 15.7%, but has an actual speedup of 10.7%.

The cost estimator predicted a zero (or close to zero) speedup for all the configurations with an odd number of MPI tasks, except for 15. Upon examining RFCTH's application communication patterns for these configurations, we found the MPI tasks are embedded into a one dimensional array where each task first com-

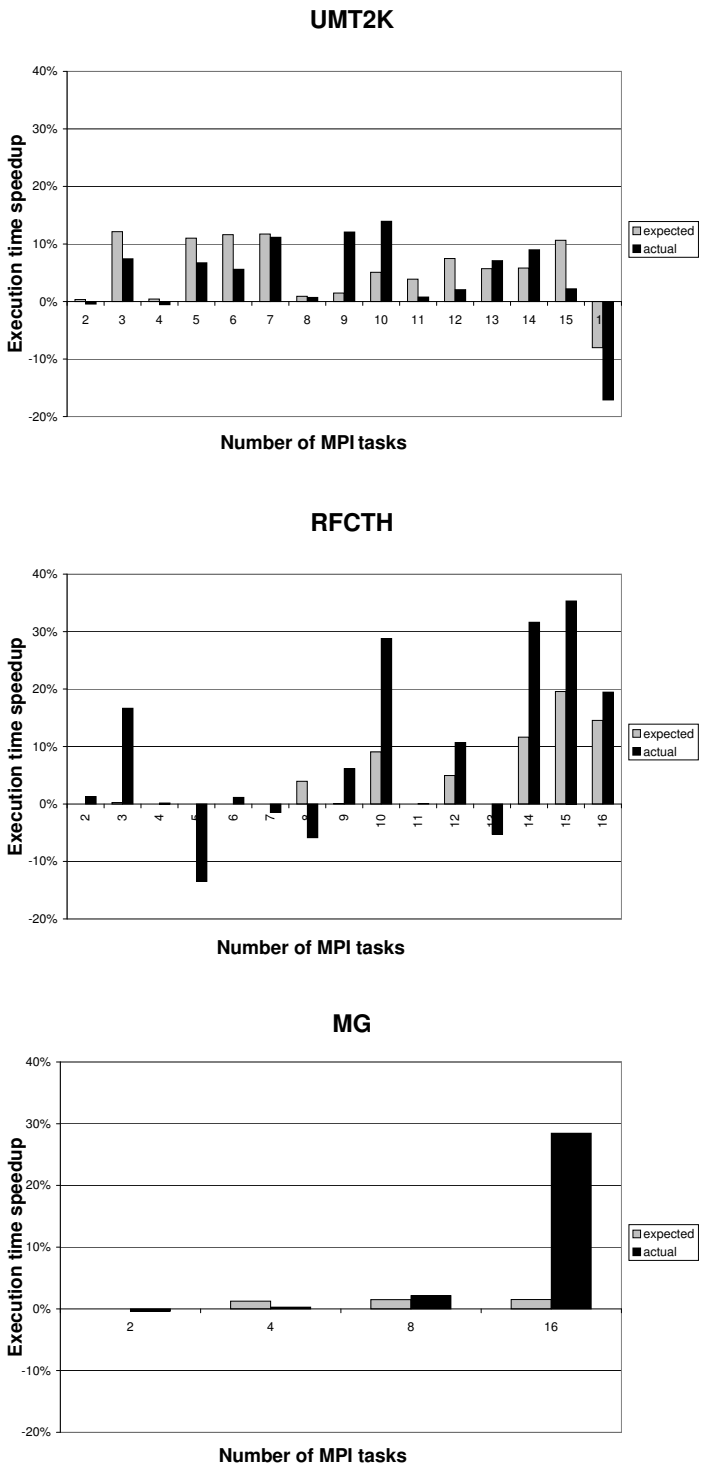


Figure 7. Cost algorithm evaluation

municates to its left neighbor and then communicates with its right neighbor. Figure 8(a) illustrates the embedding for five MPI tasks. The number of bytes communicated between neighbors are almost identical, and the difference between the two mappings is that the default mapping mapped MPI tasks together from left-to-right and the greedy mapping mapped MPI tasks together from right-to-left. The cost estimator predicts that there will be little or no difference between the two mappings, as indicated with the estimated speedup bar being zero or very close to zero.

We studied this benchmark's detailed communication patterns. The application uses blocking sends and nonblocking receives for large messages. Our manual inspection determined that the order of

the messages and the subtle interaction of the blocking/ nonblocking communication semantics may have a profound impact on performance. Because our application communication patterns do not take into account the order of messages or blocking/nonblocking communication semantics, we can not effectively model these configurations' communication behavior. We plan, in future work, to explore extending the model to incorporate message order and communication semantics.

All the other configurations for RFCTH (4, 6, 8, 10, 12, 14, 15, and 16 tasks) embed the MPI tasks into a two or three dimensional array, and our mapping algorithm is able to find mappings that perform as well or better than the default mapping.

The bottom bar graph in Figure 7 evaluates the cost estimator for MG. The cost estimator is predictable for three of the four configurations that have a non zero estimated speedup. The cost estimator has good accuracy for three of the four configurations, only the 16 task configuration has a large difference between estimated and actual speedups.

We studied the detailed communication patterns of MG. This application uses blocking sends and nonblocking receives for large messages. As with RFCTH, our manual inspection determined that the order of the messages and the subtle interaction of the blocking/nonblocking communication semantics have a profound impact on performance. Because our application communication patterns do not model the order of messages or communication semantics, we can not effectively model these configurations' communication behavior. We plan, in future work, to explore extending the model to incorporate message order and message semantics.

7.2 Mapping Algorithm Evaluation

The bar chart in Figure 9 illustrates the difference in execution time between the greedy and default mappings. A higher bar means that the greedy mapping speeds up the execution for a given configuration of the benchmark. The bar chart includes only the greedy mappings which have an estimated speedup value greater than zero[5].

The bars are grouped by benchmark. For the readers convenience, we have included an average bar for each benchmark and an average bar across all the benchmarks. The last bar illustrates a 10% average improvement across all the benchmarks if the greedy mapping is used, and the bar for RFCTH with 15 tasks has the highest actual speed up of 35.3%.

The bar chart illustrates that a slowdown occurs in only three out of the twenty-five configurations. The first two slowdowns for the 2 and 4 tasks of UMT2K are less than one percent, and fall within measurement error. The third slowdown is of 5.9% occurred for the 8 task run of RFCTH and was discussed in the previous subsection.

7.3 Discussion

Real applications have complex application communication patterns that are a function of the number of MPI tasks that are used. In Section 7.1, we described that RFCTH used application communication patterns that modeled one, two and three dimensional communication grids. Furthermore, UMT2K's application communication patterns differ significantly when the number of MPI tasks differs. When application communication patterns are a function of a command line option, it is difficult for the user to determine the best MPI task to processor mapping. Therefore automatic techniques are critical. This paper presents a model that allows an automatic approach to determine a better mapping of MPI tasks to processors.

The goal of modeling a system is to simplify the system's underlying complexity by ignoring some details while retaining other details to capture the system's essence. We model an application's communication patterns by capturing the number of messages of a particular size that are communicated between two tasks, but ignore the order in which the messages occur and ignore blocking/nonblocking communication semantics. In all three benchmarks, we have determined that message order and blocking/nonblocking communication semantics can have a significant impact on the cost estimator's accuracy. We plan, in future work, to explore extending the ACP model to incorporate message order and message semantics.

[5] Estimated speedup is computed in Section 7.1.

8. Related Work

Orduna et al [15] argue that cluster-based platforms are cost-effective for high-performance computing, and identify the interconnection network as the system bottleneck. We both have the same goal to develop task mapping techniques to reduce communication costs. We both model the traffic generated by the application and model network resources. Our work differs from their work in that they do not model specific messages, only the aggregate bytes sent between two tasks, and they do not model the on-chip memory hierarchy that allow multi-cores and multiple threads to communicate. In addition, we restrict our communication topology to be a tree, whereas they can handle an arbitrary graph [14].

There is a significant body of work on modeling communication between tasks in parallel programs [3, 8, 20, 9, 4, 11]. Most of these models are designed to analyze parallel algorithms, and typically contain a small number of parameters that abstract the communication on the machine such that machine specific features are suppressed. Most notably, the LogP model [9] and its derivatives [4, 11] intentionally remove the network topology and the routing algorithm from the model in order to to characterize an algorithm on a large class of machines. We showed that the mapping to the hardware communication topology has a considerable effect on performance, and therefore, for specific applications it is beneficial to model the network topology. Unlike the PRAM model [10] and its derivatives, our model does not account for computation. Integrating our communication model with other such models is part of our future work.

Träff [18] presents a graph embedding algorithm that optimizes the MPI communication by matching the application communication patterns to the topology using the MPI virtual topology mechanism. His study focuses on the performance of the embedding algorithm and requires the user to specify the application communication patterns and to code the virtual topology in the application. Our technique of collecting application communication patterns and estimating the communication cost is orthogonal to his embedding algorithm, and in fact can be used as input to direct the graph embedding decisions.

Pant and Jafri [16] present two complementary approaches, which extend the MPICH implementation of MPI, to reduce the communication cost of an MPI application that runs on a cluster of machines. Their topology consists of slow wide-area links that interconnect clusters and faster links to interconnect processors within a cluster. They use a profile guided optimization approach to map MPI tasks to processors to reduce the cost of point to point communication. They also replace sets of communications with collective operations (e.g. allreduce or broadcast) to minimize the traffic on the slow inter-cluster links, using topology information.

Freitag et al [12] use repeatability to infer message sizes and change the MPI library to take advantage of the extra knowledge to reduce the amount of time spent on the rendezvous protocol. Our method does not change the MPI implementation in order to reduce the amount of overhead.

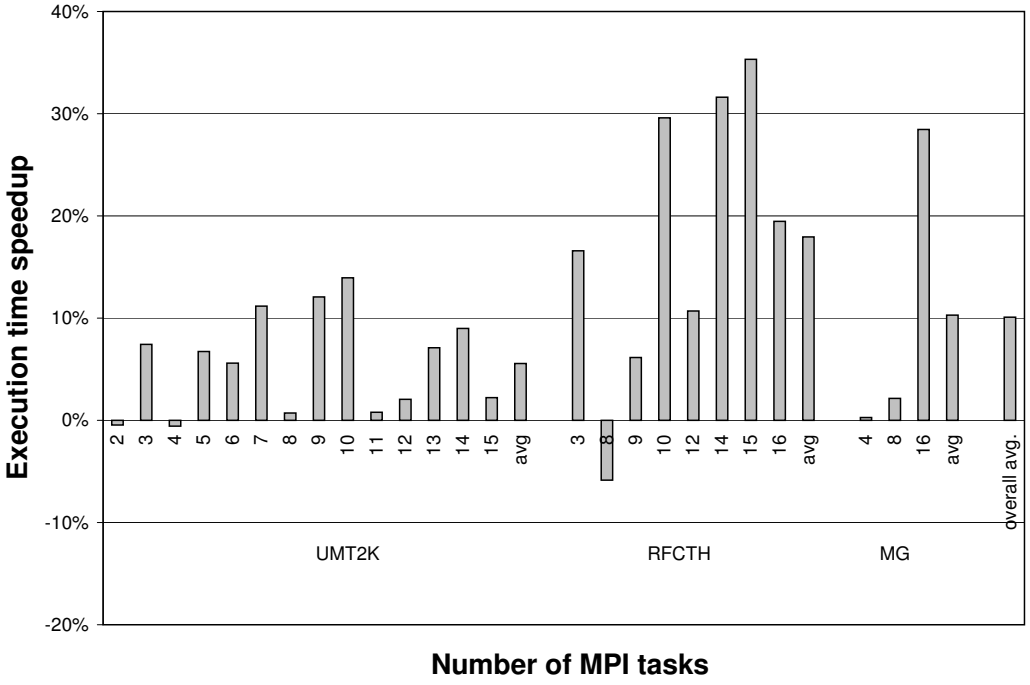


Figure 9. Greedy mapping speedup over default.

9. Future Work

The model presented in this paper strives to balance simplicity with sufficient details to adequately predict an application's communication behavior for a given execution environment.

There are some anomalies in the actual performance that we want to investigate further. In particular, RFCTH had runs where the cost estimator predicted no change while some of the actual runs achieve a performance improvement, while others showed a performance degradation. Part of this investigation will determine what needs to be enhanced: the model, the cost estimator, or both.

Currently, the cost estimator does not address applications with non-concurrent communication. There are several directions in which we can extend the cost estimator to allow handling more complex application communication patterns and we plan to evaluate them. Whether a more accurate model is needed is an interesting open question.

The default mapping is a random point in the mapping space. While it made sense to evaluate against the mapping programmers commonly use, we plan to explore the mapping space with an algorithm that takes a number of random mappings and evaluates them using the cost estimator to determine the spread of performance between a best and a worst mapping.

We are in the process of extending the model from our current ethernet network configuration of Xserve machines to a cluster of Power5 machines connected via a high bandwidth crossbar switch. As part of this environment, the Power5 processor contains two hardware threads, which requires modeling a trade-off between computation and communication.

10. Conclusions

MPI is a commonly used paradigm for parallel programming. Typically, programmers optimize application-related aspects, such as the algorithm and generic communication, rather than optimize for the execution environment. In this paper we have shown that there is significant benefit to be gained by optimizing for the execution environment. Specifically, mapping tasks to processors significantly improves performance when taking into account the application communication pattern and the communication capabilities of the underlying hardware. In keeping with the execution-environment-independent philosophy of MPI, this work is part of our larger Continuous Program Optimization (CPO) environment whereby automatic agents perform optimizations [7]. The techniques we described in the paper are amenable to that environment, thus maintaining MPI-execution independence.

After demonstrating that mapping tasks to processors in an MPI program is a critical decision that significantly impacts performance, we described how to model the application's communication pattern and how to construct a simple model of the hardware communication topology. We presented a simple mapping algorithm that uses a greedy heuristic to map MPI tasks to processors. We then described a cost estimator that takes the hardware communication topology, the application communication pattern, and a mapping from MPI tasks to processors and estimates the communication cost. Using both the cost estimator and the mapping algorithm on multiple configurations of three benchmarks, we demonstrate that the cost estimator is a good predictor of performance and up to a 35% improvement in performance over the default mapping.

Acknowledgements

This work was supported by Defense Advanced Research Project Agency Contract NBCH30390004. The authors would like to thank Evelyn Duesterwald for her participation in early discussions that helped to identify this problem. We would like to thank V. Rajan for his discussions on the complexity of finding an optimal mapping.

References

[1] Metis library. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[2] Rf-cth. http://www.cs.sandia.gov/web9232/cth/index.html.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in PRAM computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, June 1989.

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederikson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, Mar. 1994.

[6] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 50(2/3), March 2006.

[8] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony in the PRAM model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, June 1989.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1993.

[10] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[11] M. I. Frank, A. Agarwal, and M. K. Vernon. LoPC: Modeling contention in parallel algorithms. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 276–287, June 1997.

[12] F. Freitag, J. Caubet, M. Farrera, T. Cortes, and J. Labarta. Exploring the predictability of MPI messages. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, April 2003.

[13] E. Hertel, J. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In *19th International Symposium on Shock Waves*, pages 377–382, Marseille, France, July 1993.

[14] J. Orduna, V. Arnau, A. Ruiz, R. Valero, and J. Duato. On the design of communication-aware task scheduling strategies for heterogeneous systems. In *2000 International Conference on Parallel Processing (ICPP'00)*, pages 391–398, 2000.

[15] J. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. *International Conference on Parallel Processing Worksphops (ICPPW'01)*, 00:0349, 2001.

[16] A. Pant and H. Jafri. Communicating efficiently on cluster based grids with MPICH-VMI. In *Cluster 2004*, Sept. 2004.

[17] Q. Snell, A. Mikler, and J. Gustafson. Netpipe: A network protocol independent performace evaluator, 1996.

[18] J. L. Träff. Implementing the MPI process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[19] The UMT benchmark code. http://www.llnl.gov/asci/purple/-benchmarks/limited/umt.

[20] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.