Interprocessor Communication with Limited Memory

Ali Pinar, *Member*, *IEEE Computer Society*, and Bruce Hendrickson, *Member*, *IEEE Computer Society*

Abstract—Many parallel applications require periodic redistribution of workloads and associated data. In a distributed memory computer, this redistribution can be difficult if limited memory is available for receiving messages. We propose a model for optimizing the exchange of messages under such circumstances which we call the *minimum phase remapping problem*. We first show that the problem is NP-Complete, and then analyze several methodologies for addressing it. First, we show how the problem can be phrased as an instance of multicommodity flow. Next, we study a continuous approximation to the problem. We show that this continuous approximation has a solution which requires at most two more phases than the optimal discrete solution, but the question of how to consistently obtain a good discrete solution from the continuous problem remains open. We also devise a simple and practical approximation algorithm for the problem with a bound of 1.5 times the optimal number of phases. We also present an empirical study of variations of our algorithms which indicate that our approaches are quite practical.

Index Terms—Interprocessor communication, dynamic load balancing, data migration, scheduling, NP-completeness, approximation algorithms.

1 Introduction

In many parallel computations, the workload needs to be periodically redistributed among the processors. When computational work varies over time, the tasks must be reassigned to keep the workload balanced. On a distributed memory computer, this generally requires that data structures associated with the computations be transferred between processors. Many examples of this phenomena occur in scientific computing, including adaptive mesh refinement, particle simulations with short or long-range forces, state-dependent physics models, and multiphysics or multiphase simulations. For many such simulations, the limiting resource is not computation but memory. Important examples include differential equation solvers using adaptive meshes, and large-scale particle simulations. Scientists commonly choose to use the minimum number of processors upon which a particular simulation can fit, or they choose a problem size which fills the memory of a particular number of processors. However, the dynamic memory requirements of such applications make such targeting difficult.

A number of algorithms and software tools have been developed to repartition the work among processors (see, for example, [1], [2] and references therein). However, the mechanics of actually moving large amounts of data has received much less attention. When the processors have sufficient memory, the simplest way to transmit the data is quite effective. Each processor can execute the following steps:

1. Allocate space for my incoming data.
2. Post asynchronous receives for my incoming data.
3. Barrier.
4. Send all my outgoing data.
5. Free up space consumed by my outgoing data.
6. Wait for all my incoming data to arrive.

The barrier in Step 3 ensures that no messages arrive until the processor is ready to receive them, so no buffering is needed.

Unfortunately, this protocol can fail when memory is limited. It requires a processor to have sufficient memory to hold both the outgoing and the incoming data simultaneously, since incoming messages can arrive before space for outgoing data is freed. An alternative way to view this issue is that, for a period of time, the data being transferred consumes space on both the sending and receiving processors. A protocol that alleviates this problem is desirable for three reasons. First, since many scientific calculations are memory limited, reserving space for this communication operation limits the size of the calculations that can be performed. Second, the amount of memory required by this protocol is unpredictable because the data remapping requirements depend upon the computation. Thus, one is forced to set aside a conservative amount of space, which is likely to be wasteful. Third, a general purpose tool for dynamic load balancing should be robust in the presence of limited memory. On the hopefully infrequent occasions when memory limitations prohibit direct transfers, a good tool should apply an alternative strategy instead of exiting. The construction of such a tool (Zoltan [3]) inspired our interest in this problem. The desire to impact Zoltan has a number of implications for this research. We are interested in algorithms with attractive practical performance, not just asymptotic bounds on

- *A. Pinar is with the Computational Research Division of Lawrence Berkeley National Laboratory, One Cyclotron Road, MS 50F, Berkeley, CA 94720. E-mail: apinar@lbl.gov.*
- *B. Hendrickson is with the Discrete Algorithms and Math Department, Sandia National Laboratories, Albuquerque, NM 87185-1111. E-mail: bah@cs.sandia.gov.*

Manuscript received 12 May 2003; accepted 29 Oct. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0074-0503.

1045-9219/04/$20.00 © 2004 IEEE Published by the IEEE Computer Society

worst-case behavior. We also need algorithms that are straightforward and efficient to implement.

To address the deficiencies of the standard protocol, we propose a simple modification. Instead of sending all of the data at once, we will send it in phases. After each phase, processors can free up the memory of the data they have sent. That memory is now available for the next communication phase. Each phase adds an overhead due to the latency costs, thus it is important to limit the total number of phases, which is an approximation to minimize total remapping time. The results in Section 7 will confirm that the overall communication time is strongly effected by the number of phases.

More formally, consider a set of P processors. The amount of data that needs to be communicated between processors is a *transfer request*. We will assume that the request is *feasible*, that is, the end result of satisfying the transfer request does not violate any processor's memory constraints. We will let T_{ij} denote the total volume of data to be transferred from processor i to processor j. We now wish to perform the requested transfer in a sequence of phases. Let t_{ij}^l denote the volume of data transfer from processor i to processor j in phase l, and let A_i^l be the memory available to processor i at the beginning of phase l. We will also use R_i^l and S_i^l to denote the total volume of data received and sent by processor i in phase l (i.e., $R_i^l = \sum_{j=1}^{P} t_{ji}^l$ and $S_i^l = \sum_{j=1}^{P} t_{ij}^l$). At each step, the finite-memory constraint requires that $R_i^l \leq A_i^l$ for $i = 1, 2, \ldots, P$. The available memory after each phase can be computed as $A_i^{l+1} = A_i^l + S_i^l - R_i^l$. Our objective is to find a schedule of transfers that obeys the memory constraint and satisfies the transfer request in a minimal number of phases. We will call this the *minimum phase remapping problem*.

In Section 2, we show that the problem of determining whether a given transfer can be completed in a specified number of phases is NP-Complete. The remainder of this paper focuses on formulations and approximation algorithms that could be used in practice. In Section 3, we present a reduction of our problem to multicommodity flow. We present a continuous relaxation of the problem in Section 4 and a practical approximation algorithm in Section 5. Although the emphasis of this paper is a theoretical analysis, we discuss some practical issues associated with the application of our techniques in Section 6 and results of empirical studies in Section 7. Earlier versions of some portions of this paper have appeared in conference proceedings [4], [5].

Despite its practical importance, the problem of efficient data transfers has not been well-studied. Some standard collective communication operations can be implemented in ways that limit memory usage, but the general problem we consider seems to be new. Cypher and Konstantinidou designed memory efficient message passing protocols [6]. However, their work addressed exchange of tokens as opposed to variable sized messages, and they did not explicitly consider the effect of finite memory in the processors. Their work conceptually divides a process into communication and application processes. Communication processes receive unit-size messages and copy them to application processes. It is assumed that application processes have enough memory, and the goal is to limit the memory requirement of the communication processes. Very recently, Hall et al. studied the problem for large storage systems [7]. They assume the total time for subdividing a message and sending subdivided messages is about the same as sending the entire message. So, they study the problem on unit-size messages. This assumption might hold for huge volumes of data to be transferred, as in the case of reorganizing a database, however, it is far from being true for dynamic load balancing applications due to very high message setup times. Their model also restricts each processor to send and receive only one message in a phase.

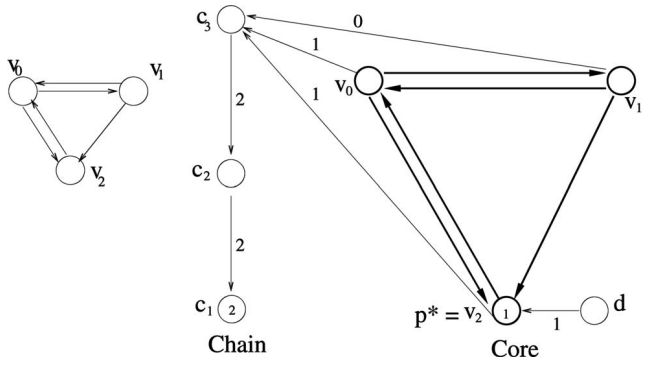

Fig. 1. Construction for NP-Completeness proof.

2 Complexity

In this section, we show that determining whether a given transfer can be completed in a specified number of phases is NP-Complete. Our proof uses a reduction from the Hamiltonian Circuit problem, which is known to be NP-Complete [8]. Recall that a Hamiltonian Circuit is a cycle in a graph that visits each vertex once. Given an instance of the Hamiltonian Circuit problem, the basic idea of our reduction is to construct an instance of the data transfer problem in which there is but a single unit of usable memory. This unit is a *token* that is passed between processors, and possession of the token allows a processor to receive data in the next phase. A solution to the data remapping problem occurs if and only if the token can be passed in a cycle among all the processors, which implies the existence of a Hamiltonian Circuit. To see how this can be done, consider the Hamiltonian Circuit problem posed in the left portion of Fig. 1. From this instance, we construct the data remapping problem in the right portion of the figure. The data remapping problem contains the original graph as its *core* (represented in the figure with dark lines) after replacing vertices with processors and replacing edges with unit-volume data transfers. It also contains a *chain* of processors to the left. The bottom processor in this chain has free memory that will percolate upwards with each phase, finally allowing all the data transfers to be completed. Given a graph $G = (V, E)$, we construct a data remapping problem as follows:

- For each vertex v_i in V there is a processor p_i. We refer to these processors as core processors. Each edge of E is a unit-volume transfer.

- Add a chain of $|V|$ processors $c_1, \ldots, c_{|V|}$ along with transfer requests from c_{i+1} to c_i for $i = 1, 2, \ldots |V| - 1$, each with volume $|E| - |V|$.
- Add a transfer request from each core processor p_i to the top of the chain $c_{|V|}$. This transfer has volume equal to one less than the in-degree of v_i in G.
- Add a dummy processor d and a unit-weight transfer connecting d to an arbitrary processor p^* in the core.
- Give $|E| - |V|$ units of free memory to processor c_1 and 1 unit of free memory to p^*. No other processor has free memory.

Consider what happens as the data remapping occurs. In the first phase, c_2 will send its data to c_1, moving the free memory one step up in the chain. After $|V| - 1$ phases, this free memory will have arrived at $c_{|V|}$, the top of the chain. Meanwhile, the token, which started at p^*, will have meandered about, enabling some data to be transferred. In phase $|V|$, processor $c_{|V|}$ has enough memory to receive all it needs from the core processors. During this phase, the token can take one more step. The messages sent to $c_{|V|}$ free up memory in the core processors. Specifically, at the end of phase $|V|$, each core processor p_i has indegree$(p_i) - 1$ units of free memory (one processor has an additional unit due to the token). In phase $|V| + 1$, core processor p_i can now receive all it needs, minus 1. The transfers to p_i can be completed in this phase if and only if one of the data transfers to p_i has previously been handled by the token. But, the only way for the token to visit all the core processors in $|V|$ phases is to complete a Hamiltonian Circuit of the core graph. Note that the token must end up where it started, at processor p^*, to enable the transfer from d to occur at phase $|V| + 1$. This argument leads to the following result.

Theorem 1. *Determining whether an instance of the data remapping problem can complete in a specified number of phases is NP-Complete.*

Proof. Given an instance of the Hamiltonian Circuit problem $G = (V, E)$, construct a data remapping problem as described above. As sketched above, the data remapping problem finishes in $|V| + 1$ phases if the core graph has a Hamiltonian Circuit. If the core graph does not have a Hamiltonian Circuit, then one of its processors will not have been visited by the token by the end of phase $|V|$. That unvisited processor, p_i, still needs to receive indegree(v_i) data, but has only indegree $(v_i) - 1$ units of available memory, so the data transfers cannot complete in $|V| + 1$ phases. The construction of the data remapping problem is polynomial, so we can conclude that the data remapping problem is NP-Hard. A given solution can be verified in polynomial time, thus the problem is in NP. □

3 Multicommodity Flow Formulation

In this section, we present a multicommodity flow (MCF) formulation to determine whether a given transfer can complete in a specified number of phases [9]. Once we can solve the decision problem, the number of phases in an optimal solution can be determined using parametric search. This formulation enables use of MCF technology to solve the minimum phase data remapping problem optimally. This might be helpful for three reasons. First, some MCF problems can be solved relatively quickly, despite their intractability in the general case. Second, the continuous version of the MCF problem can be solved in polynomial time and the solution can be used as a heuristic for the integer problem. Finally, MCF solvers will find an optimal solution if runtime is not an issue.

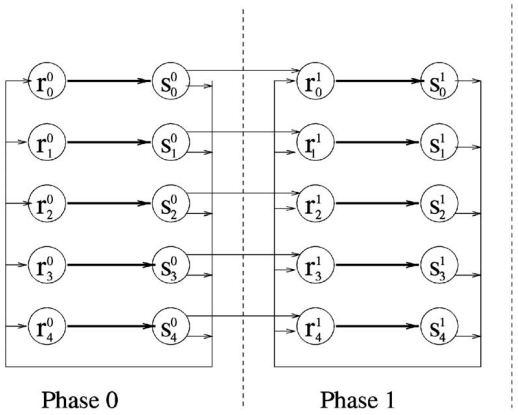

Fig. 2. MCF graph for five processors and two phases.

In our MCF formulation, each processor corresponds to a commodity. Let P be the number of processors and L be the number of phases. We must decide if a remapping can complete in L phases. As depicted in Fig. 2, our MCF graph contains a sequence of components, one for each phase. Each component allows for the communication that occurs in the corresponding phase.

The MCF graph $G = (V, E)$ has $2PL$ vertices. Each processor is represented by $2L$ vertices: two processors (one sender and one receiver) at each phase. We will use r_i^l and s_i^l to denote receiver and sender, respectively, for processor i in phase l. A sender vertex of the first phase is the source of a commodity with volume equal to the total volume of the data originally stored by this processor. A receiver vertex in the last phase is a destination for a set of commodities that corresponds to data to be stored by this processor after remapping is complete.

In the MCF graph, there is an edge from r_i^l to s_i^l for $l = 1, \ldots, L$ and $i = 1, \ldots, P$. The capacity of an edge is equal to the total memory on the respective processor. There are also edges from each sender vertex s_i^l to all other receiver vertices r_j^l in the same phase to enable data exchange between any pair of processors in a phase. These edges have infinite capacities.

With this construction, all processors first receive the data in a phase and then send their messages. This corresponds to first allocating space for the data to be received and then sending the outgoing data. The edges from receivers to senders within a phase guarantee that there is sufficient space to allocate memory for the incoming data before releasing the space for the data being shipped out, so that the memory constraints are guaranteed to be satisfied.

Finally, there is an edge (with infinite capacity) from each sender s_i^l to the receiver in the next phase r_i^l for

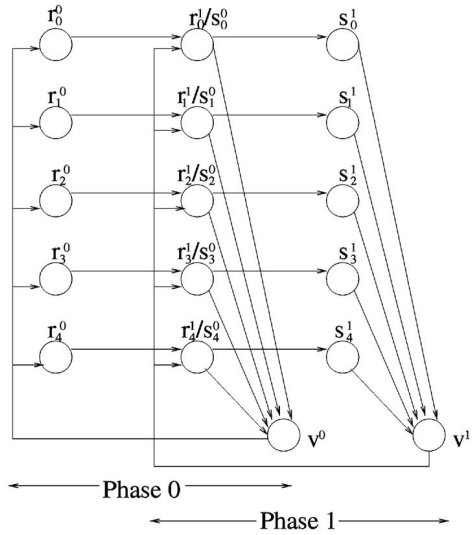

Fig. 3. MCF graph after reduction.

$l = 1, \ldots, L-1$. The flow on these edges corresponds to data that is already in the memory of a processor at the beginning of a phase. The graph for $P = 5$ and $L = 2$ is depicted in Fig. 2.

Theorem 2. *There exists a solution to the remapping problem if and only if there exists a solution to the MCF formulation.*

Proof. We can replace a data transfer from processor i to processor j in phase l with flow on edge (s_i^l, r_j^l) of equal volume. As argued above, memory constraints on the processors are satisfied if and only if the capacity constraints on the edges are satisfied in G. Thus, the feasibility of one solution implies the feasibility of the other. □

In this formulation, the number of commodities is equal to the number of processors, and the graph has $2PL$ vertices and P^2L edges. The number of vertices and edges can be reduced for a more efficient formulation. First, we can replace the crossbar between senders and receivers in a phase l with a vertex v^l and edges from all senders of phase l to v^l and edges from v^l to all receivers of phase l. Second, we can merge the senders of phase l with receivers of phase $l+1$. The graph after these reductions is depicted in Fig. 3. This improved formulation has $PL + L + P$ vertices and $(3L+1)P$ edges.

4 Continuous Relaxation

Although the multicommodity flow formulation from Section 3 provides a methodology for solving instances of the minimum phase remapping problem, runtime can still be exponential in the problem size. In this section, we describe an efficient solution for an approximation to the remapping problem. In the approximation, integral constraints on the volume of data transfers are relaxed to allow continuous values. Naturally, the volume of transfer between two processors in a phase must be an integer. But, integer solutions near the continuous ones can be used as heuristics. Note that the unit of data transfer is only a byte, whereas the volume of data being transferred is often on the order of megabytes. So, conversion from a continuous solution to an integer solution will often be a small perturbation, and so heuristics based upon this idea may be generally effective. However, bad cases for this heuristic exist, as discussed at the end of this section.

As defined in the introduction, T_{ij} denotes the total volume of data to be communicated from processor i to processor j, and t_{ij}^l denotes the volume of data transferred from processor i to processor j in phase l. The memory available to processor i at the beginning of phase l is denoted by A_i^l. We also use R_i and S_i to denote the total volume of data received and sent by processor i during remapping.

Let T be the total volume of data to be transfered, and M be the total volume of available memory in the system, then $L = \lceil \frac{T}{M} \rceil$ is a lower bound on the number of phases. We will divide each message into L equal pieces, i.e., $t_{ij}^0 = t_{ij}^1 = \ldots = t_{ij}^{L-1} = \frac{T_{ij}}{L}$ and send a piece at each phase. If the memory constraints are satisfied, then the data transfers will complete in precisely L phases. However, there is no guarantee that memory constraints will not be violated. As a solution to this problem, we will use preprocessing and postprocessing phases to ensure feasibility of the phases in between.

Lemma 1. *If the following conditions are satisfied, the continuous version of the remapping problem can be completed in $L = \lceil \frac{T}{M} \rceil$ phases.*

1. $S_i = R_i$ *for all processors.*
2. $A_i^0 \geq \frac{R_i}{L}$.

Proof. At each phase, processor i will receive $\frac{R_i}{L}$ units of data. By the second condition, each processor has sufficient memory for the first phase. By the first condition, each processor ships out $\frac{S_i}{L} = \frac{R_i}{L}$ units of data at each phase, which frees up sufficient memory for the next phase. □

Lemma 2. *A solution for a continuous version of the data remapping problem for transfer request $\mathcal{R}$ can be performed via the following three steps:*

1. *One preprocessing phase.*
2. *A new transfer request $\mathcal{R}'$, where $S_i = R_i$ and $A_i^0 \geq \frac{R_i}{L}$.*
3. *One postprocessing phase.*

Proof. In the preprocessing phase we will reorganize the data to satisfy conditions 1 and 2 from Lemma 1, and define a new mapping of the data. After the new mapping is complete, a single postprocessing phase will be sufficient to get all of the data to the correct processor.

In the preprocessing step, all processors i with $R_i < S_i$ will transfer some of their outgoing data to processors j in which $R_j > S_j$, so that in subsequent phases $R_i = S_i$. Note that, if the transfer request is feasible, then $R_j - S_j \leq A_j^0$. Thus, this rearrangement can be completed in a single phase.

Next, as second part of the preprocessing step, all processors i with $A_i < \frac{R_i}{L}$ will transfer some of their outgoing data to processors j with $A_j > \frac{R_j}{L}$. To avoid disturbing the first property, the sending processors will also pass equal amounts of receiving assignment. Once

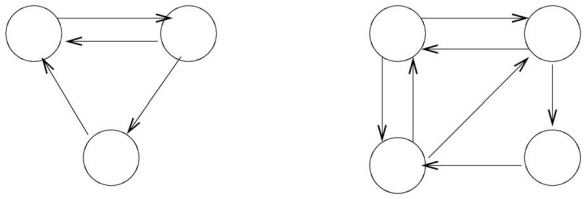

Fig. 4. Catastrophic instance for continuous relaxation.

again, this step can be completed in one phase, since by construction, the receiving processors have sufficient space.

Notice that the actual data being transferred is irrelevant—we are just trying to balance the numbers—so a send and receive operation can cancel each other. This enables merging of the two steps above into one phase.

After the new transfer request $\mathcal{R}'$ is realized, we must correct for the transfer of receiving assignments. This correction is the purpose of the postprocessing phase. Under the transfer of receiving assignments, each processor is either a sender or a receiver of such assignments. So, during postprocessing, each processor will either receive or send data, but not both. Since the initial remapping is feasible, each processor has enough memory for the data to be received, thus the postprocessing can be completed in one phase. □

The complexity of constructing the solution for the preprocessing phase is linear in the number of processors. To see this, divide the processors into two lists: those with $R_i < S_i$ and those with $R_j > S_j$. Now, step through the lists together, transferring sending responsibility from a processor in the i list to one in the j list. Each transfer balances R_i and S_i for a processor in one of the lists. The same can be applied to balance initial available memories. Notice that the preprocessing step uniquely describes the postprocessing phase, and remapping for $\mathcal{R}'$ is straightforward.

Theorem 3. *Given a transfer request $\mathcal{R}$, the continuous version of the data remapping problem can be completed in $\lceil \frac{T}{M} \rceil + 2$ phases.*

Proof. By Lemma 2, $\mathcal{R}$ can be completed by pre and postprocessing steps, along with a transfer request $\mathcal{R}'$ satisfying conditions of Lemma 1. Notice that the total volume of data to be transferred, T' in $\mathcal{R}'$ is no greater than T in $\mathcal{R}$, and the total available memory in the system does not change: $M = M'$. Hence, by Lemma 1, $\mathcal{R}'$ can be completed in $\lceil \frac{T'}{M} \rceil \leq \lceil \frac{T}{M} \rceil$ phases. Together with one preprocessing and one postprocessing phases, remapping can be completed in $\lceil \frac{T}{M} \rceil + 2$ phases. □

It is worth noting that a good solution for this continuous approximation may not yield a good solution for the true discrete problem. For instance, consider the example depicted in Fig. 4.

This example consists of two groups of processors, with no communication between the groups, and there is only one unit of available memory. Available memory must be possessed by each component in turn, and this requires temporarily moving some data from one component to the other to transfer the free memory, as will be discussed in more detail in the next section. In the preprocessing step described in the proof of Lemma 2, this available memory will be divided into two groups of processors, but the fractional transfers that follow give no insight into the correct way to orchestrate the data transfers for this instance. Specifically, in the continuous solution, all processors are identical, so no information is gleaned about the necessity of working on components in turn.

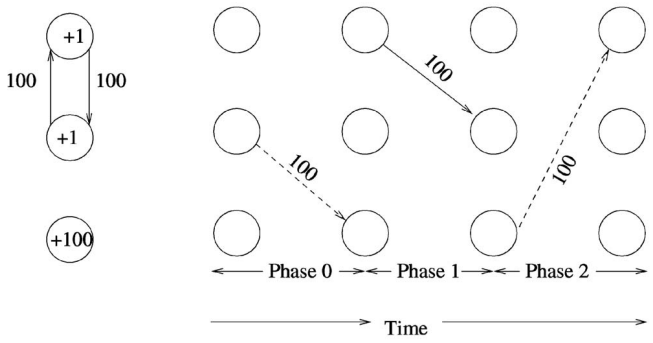

Fig. 5. Example of the utility of parking.

5 Efficient Approximation Algorithms

In this section, we describe the basics of a family of efficient algorithms that provide solutions, in which the number of phases is at most 1.5 times that of an optimal solution. The algorithm is motivated by some simple observations. First, the maximum amount of data that can be transferred in a phase is equal to the total amount of free memory in the parallel machine. Let M be the total available memory in the parallel machine, and let T be the total volume of data to be moved. Note that M does not change between phases.

Lemma 3. *The minimum number of phases in a solution is $\lceil \frac{T}{M} \rceil$.*

This bound can only be achieved only if available memory is used to receive messages at each phase. Thus, free memory is wasted if it resides on a processor that has no data to receive. Our algorithm works by redistributing free memory to processors that can use it. Equivalently, data is *parked* on a processor with free memory, which it cannot use, to free up memory on processors that can use it. We will park only data that must be transferred eventually.

5.1 Parking

Parking aims to utilize memory that would otherwise be wasted. Consider a processor that has received all its data and still has available memory. This memory cannot be utilized in subsequent phases, which decreases the total memory usable for communication, thus potentially increasing the number of phases. Instead, another processor can temporarily move some of its data to this processor to free up space for messages. An example is illustrated in Fig. 5. In this simple example, the top two processors want to exchange 100 units of data, but each has only one unit of available memory. A simplistic approach will require 100 phases. However, the third processor has 100 units of free memory. By *parking* data on this third processor (i.e., transferring free memory to another processor), the number of phases can be reduced to three.

More formally, if a processor has k units of data left to receive and m units of free memory, then it has *parking space* of $\max(0, m - k)$ units. A processor has data to park if the incoming data exceeds available memory, and the quantity

of this parkable data is $\max(0, k-m)$ units. The parkable data consists of data that eventually must be sent to another processor. Note that, if the transfer request is feasible, then a processor must send out $\max(0, k-m)$ units. Any processor that has parking space can store parkable data from another processor, thereby maximizing the amount of usable free memory. This parked data merely takes an extra step on the way to its final destination. Exploiting this observation will allow us to construct an approximation algorithm.

In our algorithm, we merely store data in a parking space, and then forward it to its correct destination when the destination processor has available memory. Note that it is inconsequential which processor owns the parked data. In other words, parking spaces are indistinguishable. What potentially effects performance is which processors shunt their data to a parking space.

Lemma 4. *It is sufficient to park data at most once to obtain an optimal solution.*

Proof. Assume there is a solution that parks some data D twice. Let p_1 and p_2 be the first and second processors on which D is parked. After data is moved from p_1 to p_2, if no other processor uses available memory at p_1, then there was never a need to move data to p_2. If another processor p_i, parks data on p_1, then we can rearrange the data movement with D staying in p_1, and p_i parking on p_2, due to indistinguishability of parking spaces. □

It is worth noting that parking is not just a heuristic, but a requirement in some cases. Consider the example in Fig. 5, modified so that there is no available memory in the top two processors. In this case, the transfer request is still feasible, but realizing the remapping requires parking.

5.2 An Approximation Algorithm

In this section, we describe an algorithm that obtains a solution with at most 1.5 times the optimal number of phases. The algorithm is quite generic and allows for a number of possible enhancements.

Algorithm 1:

- A processor receives as much data as it can in each phase (i.e., if a processor has available memory at the end of a phase, then this processor does not have any more data to receive).
- If the transfer request cannot be completed in the next phase, then park as much data as possible (i.e., park the minimum of the total parkable data and the total parking space).

Note that many details about the algorithm are unspecified: If I have more incoming data than free memory, which messages should I receive in the current phase? If several processors want to park data, but limited parking spaces are available, which should succeed? We will show below that, with any answers to these questions, the resulting algorithm generates a solution with no more than 1.5 times the optimal number of phases. Intelligent answers to these questions could be used to devise algorithms with better practical (or perhaps theoretical) performance.

Lemma 5. *The total volume of data transferred by Algorithm 1 is at most* $\lceil \frac{3T}{2} \rceil$.

Proof. Let T_p be the volume of data transferred through parking, and let T_d be the data transferred directly. Data is transferred either directly or through parking, thus $T = T_p + T_d$.

It is enough to park data once due to Lemma 4, thus parked data is moved twice, and the total volume of data moved is $2T_p + T_d = T + T_p$. Recall that, by definition of parkable data, data is parked only if it will help receiving more data in the next phase. Thus, each parked unit of data enables at least one direct transfer, the algorithm guarantees that $T_p \leq T_d$. Thus, at most half of T can be transferred through parking, i.e., $T_p \leq \frac{T}{2}$, and the total volume of data moved is $T + T_p \leq T + \frac{T}{2} = \frac{3T}{2}$. □

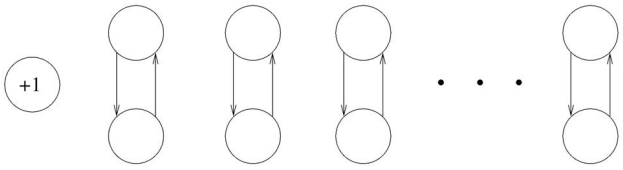

Fig. 6. Example to show the tightness of the 1.5 bound.

Theorem 4. *Algorithm 1 constructs a solution with at most* $\lceil \frac{3T}{2M} \rceil + 1$ *phases.*

Proof. The algorithm makes use of all M units of available memory until the amount of parkable data is less than the amount of parking space. It then completes in at most two additional phases, one in which some data is parked, and a final phase in which each processor has enough memory to receive all its messages. By Lemma 5, we know that the total volume of data transferred in the algorithm is at most $\lceil \frac{3T}{2} \rceil$. With M units of transfer in all but the last two phases, the process can be completed in at most $\lceil \frac{3T}{2M} \rceil + 2$ phases.

We will now decrease the bound to $\lceil \frac{3T}{2M} \rceil + 1$. Let l be the number of phases for the algorithm to complete the data remapping process. The total volume of data transferred is $(l-2)M + x$, where x is the volume of data transferred in the last two phases: $1 < x \leq 2M$. From Lemma 5, we know that

$$\left\lceil \frac{(l-2)M + x}{M} \right\rceil \leq \left\lceil \frac{3T}{2M} \right\rceil.$$

But, simple algebra reveals that

$$l - 1 \leq \left\lceil \frac{(l-2)M + x}{M} \right\rceil.$$

Combining these inequalities

$$l - 1 \leq \left\lceil \frac{3T}{2M} \right\rceil,$$

and the result follows. □

Combined with Lemma 3, Theorem 4 shows that Algorithm 1 is a $3/2$ approximation algorithm for the minimum phase remapping problem. Without a better lower bound, this value of $3/2$ is tight as illustrated by the example in Fig. 6.

This example consists of an odd number of processors P. All but one of them are organized in pairs which exchange a single unit of data. Only the unpaired processor has a single

```
While I have data to send or receive do
  Apportion available memory to senders.
    Post an asynchronous receive for each allotment.
    Inform senders their allotments and collect mine.
  for each sending allotment I am given
    Construct and send message.
    Free up the memory for data sent.
```

Fig. 7. Template of a distributed algorithm for memory-constrained interprocessor communication.

unit of available memory. The total volume of data to be moved is $T = P - 1$. The only way for a pair to exchange their data is first to park a unit elsewhere, so a total of $\frac{P-1}{2}$ units of parking are needed. Hence, the total volume of data transferred is $\frac{3(P-1)}{2} = \frac{3T}{2}$, and the number of phases is $\frac{3T}{2M}$, since $M = 1$.

6 Practical Considerations

In this section, we consider some more practical aspects of the minimum phase remapping problem, setting the stage for experimental results in the subsequent section. First, we discuss practical aspects of algorithms for scheduling the communication phases, and efficient implementations of heuristics based on the templates of the preceding sections. Then, we consider how to implement parking and, finally, we address some practical limitations of our model and how to address them.

6.1 Distributed and Centralized Heuristics

Algorithms for minimizing the number of phases can be either centralized or distributed. Centralized heuristics reduce the problem onto one processor, construct a solution on this processor, and then broadcast the solution to all processors. Reducing the whole problem onto a single processor enables the use of sophisticated algorithms to construct optimal or near optimal solutions. However, these methods invariably suffer from the gather and scatter overhead due to collecting the information on one processor and broadcasting the solution. Moreover, the remaining processors wait idle while one processor is constructing a schedule, which hinders scalability.

An alternative is a distributed method, where processors collectively produce a solution using local information. A distributed approach is more scalable, but the lack of global information is likely to lead to lower quality solutions. We present experiments that compare the performance of the two approaches in the next section. Here, we discuss how to implement a distributed scheduling algorithm efficiently. Fig. 7 presents a distributed algorithm for interprocessor communication with limited memory. Notice that there are no global operations in this algorithm; processors communicate only with their neighbors—processors they need to exchange data with. This locality is important for scalability.

The number of phases is determined by how available memory is apportioned to the processors, which is not specified in Fig. 7. Below, we describe five heuristics that rely only on local information. No extra communication is required to make apportionment decisions. Four of our heuristics are nonpreemptive, that is, we chose a processor and grant it as much memory as it requires or is available, and repeat the process while there is available memory. We experimented with the following four criteria for choosing a sender:

- *First-fit* (FF): Choose the first processor on the list of senders.
- *Max-first* (MF): Choose the processor that has the most to send.
- *Random* (RD): Choose a processor uniformly at random.
- *Fortune Wheel* (FW): Choose a processor randomly, but the probability of a processor being chosen is directly proportional to how much it has to send.

We also have a preemptive heuristic, where we grant allotments so that the maximum remaining volume to be sent by any sender is minimized. For example, if processor 1 has five units whereas processor 2 has four units to send, we grant five units of available memory as three units to processor 1 and two units to processor 2, which leaves three units to send for each processor. We refer to this heuristic as the *Min-max* (MM) heuristic. We compare practical performances of these heuristics in the experimental results section.

```
X ⟵ (Volume of remaining data)
       − (Volume of available memory).
If X ≥ 0 inform the root that I can park X units.
else inform the root that I can receive X units.
if I am the root
  Collect parking information.
  Match parkable data with parking space and
  inform processors.
Receive parking decisions from the root.
If I am receiving parkable data
  Post asynchronous receives; update send list.
else if I am parking
  Pack and send data; update send list.
Communicate with neighbors to update receive lists.
```

Fig. 8. Template for parking.

6.2 Parking

Parking helps to decrease the number of phases by utilizing available memory on processors that do not have more to receive. However, the parking operation itself can be expensive for several reasons. First, the total volume of communication increases, since data being parked are moved twice. More importantly, parking is a global operation: any processor needing to receive can park data on any other processor with available memory, and so some kind of global oversight and control is necessary.

Implementation of a parking algorithm requires three steps: making parking decisions, moving parked data, and updating send/receive information. An algorithm that follows these steps is presented in Fig. 8. This algorithm uses a root processor to make the parking decisions. Moving data is straightforward, since processors already know to whom to send and from whom to receive. Finally, we can update receive lists by communicating with the neighbor processors. As can be seen, parking is not a simple

operation, and as will be shown in Section 7, its overhead cannot always be compensated for by decrease in the number of phases.

6.3 Limitations of the Model

The preceding sections of this paper have concentrated on the amount of available memory, ignoring how that memory is organized. However, the adaptive applications that require periodic data redistribution typically make extensive use of dynamic memory management. As a consequence, the available memory may not be contiguous, which limits the immediate practical utility of our techniques. In the following discussion, we assume that the message passing software requires outgoing/incoming messages to be read from/to a single, contiguous block of memory. This is typical of current implementations of MPI.

The complexity of memory layout adds several complications to our simple theoretical model. First, outgoing data will not generally be contiguous, so space to buffer it on the sending processor will be needed. Second, if available memory is fragmented, then incoming messages can only be as large as the fragments, not the entire free memory. Once data has arrived, it may need to be copied into noncontiguous portions of an application data structure. Finally, our analysis assumed that total available memory at the end of a phase could be updated simply by the difference between the total amounts of outgoing and incoming data. Although this may be true, it says nothing about fragment sizes, which are the true, practical limit on useful memory for message passing.

Despite these caveats, our model and algorithms can sometimes be applied without modifications. Consider the case where the application controls memory and packs all data contiguously. Available memory will then consist of a single, contiguous block. If the application data structures allow for the memory to be reorganized to make outgoing message data contiguous, then no further buffering of outgoing data will be necessary. Once messages have arrived, memory can be locally reorganized to place them into the application data structures in the appropriate way, while keeping the available memory contiguous. In this way, the objections raised above are circumvented and our model and techniques are directly applicable.

However, most applications do not allow for such straightforward memory management techniques. For such applications, our theoretical model provides guidance, but not robust theoretical guarantees of performance. In Fig. 9, we sketch an algorithm that performs robustly, but without the certainty of bounds on the number of phases. The basic idea is to have each processor determine its available memory at the beginning of each phase and to restrict the sizes of incoming messages to ensure that this memory is not exceeded. No assumptions are necessary about the amount of available memory in subsequent phases. We will not report results on the performance of this heuristic, since its performance directly depends on memory segmentation.

```
While data exchange is not complete
  Determine my receive-buffer and send-buffer sizes.
  Communicate my send-buffer size.
  While there is available memory and data to receive
    Choose a set of senders who will send to me.
    Choose message sizes that senders will send to me.
    Post asynchronous receives.
    Inform senders of their allotments.
  for each sending allotment I am given
    Construct and send message.
    Free up the memory for data sent.
```

Fig. 9. Template for a robust algorithm for memory-constrained interprocessor communication.

7 Experimental Results

We have implemented the algorithms discussed in Section 6, and performed a series of empirical studies on the LBL/NERSC Alvarez Cluster, an 80 node Pentium III Myrinet cluster with two processors in each node. All methods were implemented in C, and use MPI for message passing.

The problem instances in our experiments come from the data distribution associated with load balancing the operation a particular preconditioner known as overlapped Schwartz domain decomposition. The communication involved can be described in a fairly simple way. Initially, a graph is partitioned in such a way that each processor has about the same number of vertices. Now, imagine that each processor needs to perform an operation that depends on the number of vertices it owns plus the number of vertices on other processors that are adjacent to ones it owns. These "overlapped subdomains" will not generally be well balanced. But, with a modest adjustment to the original partition, the overlapped domains can be made balanced (see [10] for details). The communication operation we are using in our experiments is that of transforming the original partition into a balanced overlapped partition.

This communication is typical of those encountered in dynamic load balancing. The communication pattern is sparse and unstructured. Note also that different initial partitions lead to different communication patterns, so it is easy to generate a set of different, but related remapping instances. In all our experiments, we used the matrix `ocean`, which has 143,437 rows and 819,186 nonzeros, with eight double-precision numbers for each mesh node.

We report our results on four sets of experiments. The objective of the first set of experiments is to determine whether or not minimizing the number of phases truly is a valid metric for reducing overall communication cost. Our second experiment studies the relative merits of distributed and centralized approaches as discussed in Section 6.1. Third, we study the performance of several different distributed heuristics, including those introduced in Section 6.1. Finally, we consider the practical merits of parking.

- *Experiment 1: Model Validation*. On the first set of experiments, we investigated the relation between the number of phases and the communication time. For scheduling methods, we implemented the *Random*, *Fortune Wheel*, and *First Fit* techniques from Section 6.1. Keeping the communication pattern the same, we reduced the amount of memory the algorithms were allowed to use, which increased the number of phases, but kept the volume of data transfer unchanged.

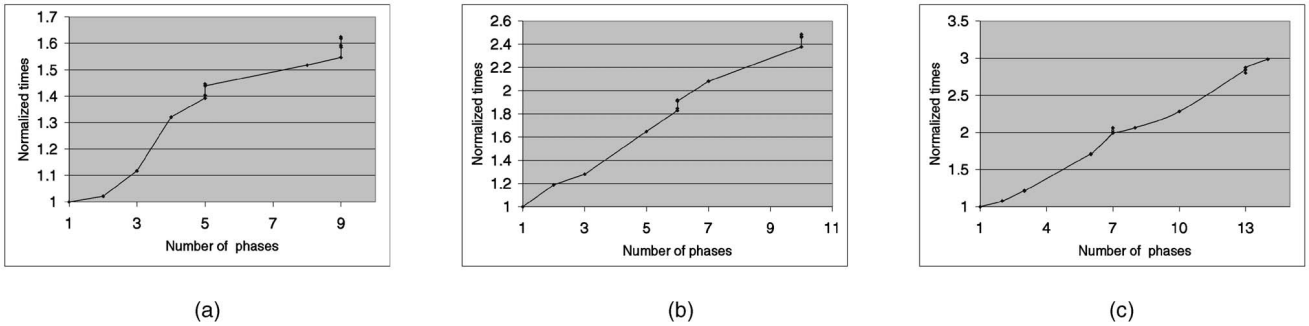

Fig. 10. Correlation between number of phases and communication times. (a) 32 processors, (b) 64 processors, and (c) 128 processors.

The results of these experiments are presented in Fig. 10. In this figure, the horizontal axis correspond to the number of phases, and the vertical axis correspond to the reorganization time normalized with respect to reorganization time without memory limitations, i.e., communication in a single phase. Each point in these graphs corresponds to a single instance in our experiments. It is important to note that decision making part in this heuristics take almost no time, thus the runtimes are essentially only the communication times. The results show that runtimes grow roughly linearly with increasing number of phases, and that the the cost of each phase is a nonnegligible fraction of the single phase communication cost (particularly for the larger numbers of processors). Thus, minimizing the number of phases is a valid objective for scheduling.

- *Experiment 2: Centralized Versus Distributed Methods*. In Fig. 11, we compare centralized and distributed methods. In this figure, times for the centralized methods are broken down to four components: "gather" is the time to collect problem information on one processor, "solve" is the time that one processor spends constructing a schedule, and "scatter" is the time to inform all processors of the schedule. We experimented with the centralized and distributed implementations of the five heuristics described in Section 6.1 on eight problem instances. To take maximal advantage of the availability of all the information on a single processor, the centralized methods used parking to minimize the number of phases. The distributed methods in this experiment did not use parking.

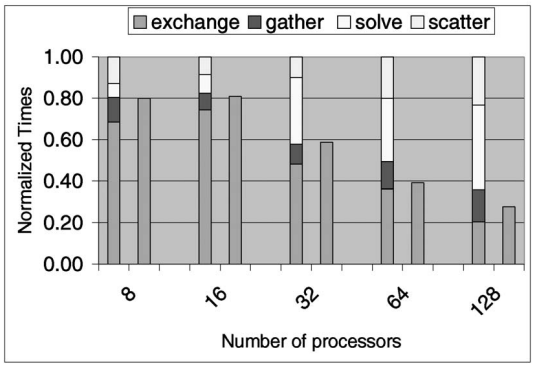

Fig. 11. Centralized and distributed methods.

For comparison purposes, times are normalized with respect to the total runtime of the centralized implementation. We have not observed significant differences in the relative performances of the different methods, so we present only the averages of normalized times in Fig. 11. As seen in the figure, the overhead for the centralized methods is quite high due to the gather and scatter time, and the serial algorithmic bottleneck. However, the centralized method is able to produce modestly better schedules. Our data indicates that distributed methods are generally more efficient. However, if the schedule is to be reused, then perhaps the significant overhead induced by the centralized method could be justified.

- *Experiment 3: Different Distributed Methods*. We have implemented and compared the five distributed heuristics introduced in Section 6.1, as well as the continuous relaxation method (CR) described in Section 4. Since we are interested in a general purpose tool, our principle concern is with the total time consumed by a collection of invocations. For this reason, we ran the algorithms on 10 problem instances, counting the number of phases in the communication schedules they produce. The results of the average number of phases for each method are displayed in Fig. 12a. It can be seen that randomized techniques (RD and FW) have the best performance, followed by the min-max technique (MM). The continuous relaxation method suffers from its pessimistic nature, since it always requires two extra phases, even for very simple problems. The first-fit heuristic (FF) suffers from the fact that communication lists are often sorted, and so all processors will try to communicate with the lowest numbered processors first. The poor performance of max-first (MF) can be attributed to natural communication patterns of dynamic load balancing. Dynamic balancing shifts load from overloaded processors to underloaded processors, thus some overloaded processors send a lot of data, but receive (almost) none. The max-first technique is likely to give priority to such processors, and so lead to poor distribution of the communication load.

Although the number of phases is our primary metric, the runtime of each phase is also important. The number of messages in a phase can effect performance due to large message setup times. Thus, schedules with fewer number of messages

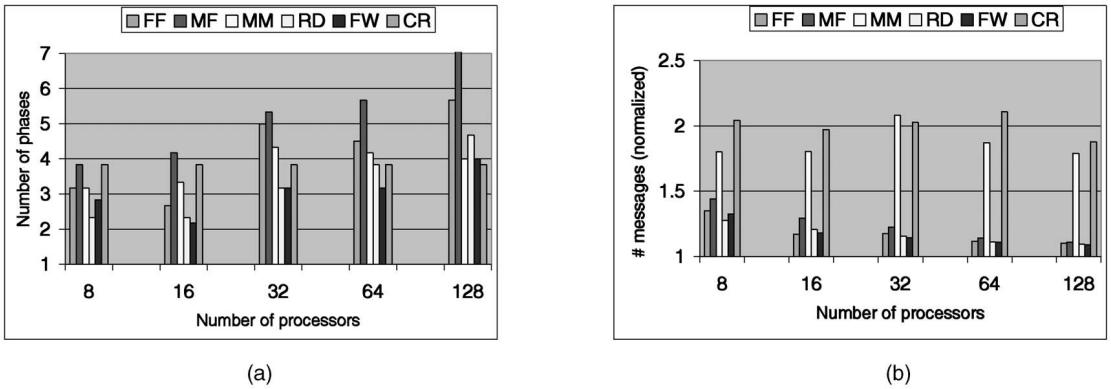

Fig. 12. Comparison of scheduling heuristics. (a) Number of phases and (b) number of messages.

are preferred. Fig. 12b compares the heuristics in terms of the number of messages. As before, numbers are computed as the average of 10 different instances, but here the results are normalized by the number of communicating pairs of processors (i.e., the number of messages without memory limitations). As expected, the number of messages is very high for the min-max and continuous relaxation heuristics. Performances of the other heuristics are very close. Recall that the continuous relaxation technique reorganizes data in pre and postprocessing phases, thus the total volume of communication is higher than what is required. We observed that, in practice, the additional data movement for this method is very high—52 percent, 46 percent, and 44 percent for 32, 64, and 128 processors, respectively. Overall, we can conclude that randomized techniques perform better than the other heuristics.

- *Experiment 4: Parking*. As discussed in Section 5, parking utilizes memory that would otherwise be wasted, potentially decreasing the number of phases. Our experimental results as presented in Fig. 13, show that in practice the decrease in number of phases due to parking can be very significant. In many cases, the number of phases can be halved by using parking.

  We have already discussed the overhead due to parking in Section 6.2. In our implementation, parking requires a centralized algorithm, and so bears the costs associated with gathering data and broadcasting an answer as discussed above. In addition, parking increases the total volume of communication since data are routed through an intermediate processor instead of being sent directly to their destination. Our experiments showed that this is not a significant problem. The total volume of communication increases by only an average of 1 percent for 32 processors, 3 percent for 64 processors, and 3 percent for 128 processors, which can easily be compensated for by the significant decrease in the number of phases. The important problem is how to limit the cost of the parking operation itself.

  To retain the benefits, but reduce the cost, we tried using parking less frequently to decrease its cost, but still be able limit the number of phases. Fig. 14 presents the results for these experiments. In this figure, P1 corresponds to parking at each phase and P2 (P3) corresponds to parking at every second (third) phase. The total available memory decreases from left to right for the problem instances. The times are normalized with respect to the times of solutions without parking. The results show that parking can decrease overall data reorganization times, especially when memory is very limited.

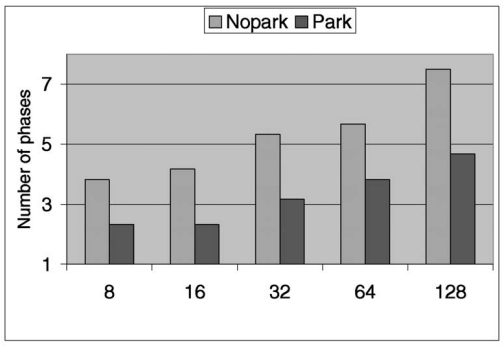

Fig. 13. Effect of parking on the number of phases.

8 Conclusion

We have addressed the problem of migrating data in the case of limited memory. When migrating data in an adaptive computation, it can happen that processors do not have enough memory to allocate space for their incoming data before they can release the space consumed by their outgoing data. In this case, the remapping operation must be decomposed into phases so that processors free up memory for the data they shipped out at end of a phase, making it available for the incoming data in the next phase.

In this paper, we studied how to complete the remapping operation in the minimum number of phases, the problem we call minimum phase remapping. We showed that the problem of determining whether a given transfer can be completed in a specified number of phases is NP-Complete. A reduction of the minimum phase remapping problem to multicommodity flow was presented. We showed how a continuous relaxation of the problem admitted a simple solution with two more phases than that of an optimal solution, but it might be difficult to obtain a good discrete solution from this continuous one. We also devised a practical

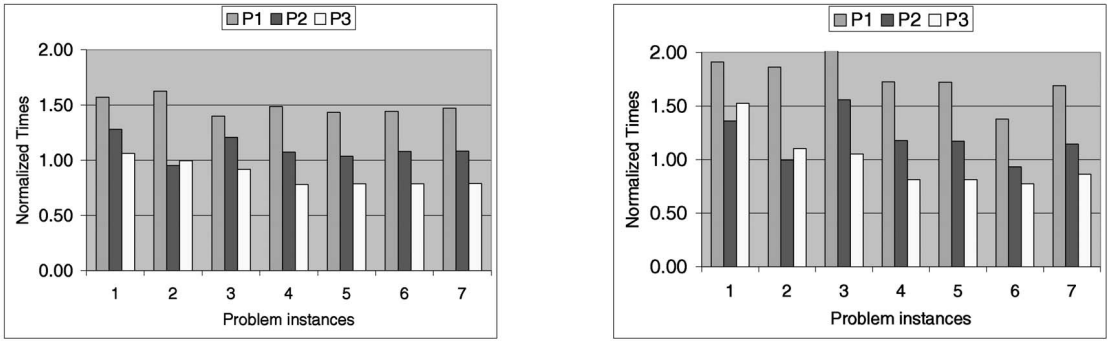

Fig. 14. Effect of parking on the number of phases.

approximation algorithm with a bound of 1.5 times the optimal solution.

We then conducted a series of empirical studies to validate the relevance of the number of phases as a metric of communication cost, and to study the performance of different versions of our algorithms. These experiments indicate that our methods can be implemented efficiently in parallel, and that multiphase remapping can be performed inexpensively.

Our interest in this problem arises from our collaborative efforts to build general purpose libraries to support complicated parallel applications. Specifically, we are interested in providing additional robustness to parallel libraries. The functionality described here is now being added into Zoltan, a public-domain dynamic load balancing tool [3].

ACKNOWLEDGMENTS

This work was funded by the Applied Mathematical Sciences progam, US Department of Energy, Office of Energy Research and Corporation, a Lockheed-Martin Company, for the US DOE under contract number DE-AC-94AL85000. The first author is also supported by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the US Department of Engergy under contract DE-AC03-76SF00098.

REFERENCES

[1] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel Distributed Computing,* vol. 7, pp. 279-301, 1989.
[2] B. Hendrickson and K. Devine, "Dynamic Load Balancing in Computational Mechanics," *Computer Methods in Applied Mechanics and Eng.,* vol. 184, nos. 2-4, pp. 485-500, 2000.
[3] K.D. Devine, B. Hendrickson, E.G. Boman, M.M. St. John, and C. Vaughan, "Zoltan: A Dynamic Load-Balancing Library for Parallel Applications—User's Guide," Technical Report SAND99-1377, Sandia Nat'l Laboratories, Albuquerque, New Mexico, http://www.cs.sandia.gov/Zoltan/, 1999.
[4] A. Pinar and B. Hendrickson, "Interprocessor Communication with Memory Constraints," *Proc. 12th ACM Symp. Parallel Algorithms and Architectures,* pp. 39-45, July 2000.
[5] A. Pinar and B. Hendrickson, "Communication Support for Adaptive Computation," *Proc. 10th SIAM Conf. Parallel Processing for Scientific Computing,* Mar. 2001.
[6] R. Cypher and S. Konstantinidou, "Bounds on the Efficiency of Message-Passing Protocols for Parallel Computers," *SIAM J. Computing,* vol. 25, no. 5, pp. 1082-1104, 1996.
[7] J. Hall, J. Hartline, A. Karlin, J. Saia, and J. Wilkes, "On Algorithms for Efficient Data Migration," *Proc. Symp. Discrete Algorithms,* 2001.
[8] R.M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, eds., pp. 85-103, New York: Plenum Press, 1972.
[9] R.K. Ahuja, R.L. Magnanti, and J.B. Orlin, *Network Flows: Theory, Algorithms and Applications.* Englewood Cliffs, N.J.: Prentice Hall, 1993.
[10] A. Pinar and B. Hendrickson, "Partitioning for Complex Objectives," *Proc. 15th Int'l Parallel and Distributed Processing Symp.,* 2001.

Ali Pinar received the BS and MS degrees in computer engineering from Bilkent University, Turkey, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He is currently working at Lawrence Berkeley National Laboratory. His research interests include combinatorial scientific computing, combinatorial algorithms, and parallel algorithms. He is a member of the IEEE Computer Society, SIAM, and ACM.

Bruce Hendrickson received degrees in math and physics from Brown Univeristy, followed by the PhD degree in computer science from Cornell University. He has been at Sandia Labs in Albuquerque, New Mexico for the past 13 years, where he is a distinguished member of technical staff and acting manager of the Discrete Algorithms and Math Department. He also has an appointment in the Computer Science Department at the University of New Mexico. He is an editor of several leading journals in scientific and parallel computing, and has helped to organize numerous international meetings. His research interests include combinatorial scientific computing, linear algebra, and parallel algorithms. He is a member of the IEEE Computer Society, SIAM, and ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.