SAND2000-1361C

# Interprocessor Communication with Memory Constraints*

RECEIVED JUN 20 2000 OSTI

Ali Pınar
Department of Computer Science
University of Illinois
Urbana, IL 61801
alipinar@cse.uiuc.edu

Bruce Hendrickson
Parallel Computing Sciences Department,
Sandia National Laboratories
Albuquerque, NM 87185–1110
bah@cs.sandia.gov

## 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 multi-commodity 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. Finally, we devise a simple and practical approximation algorithm for the problem with a bound of 1.5 times the optimal number of phases.

## 1. INTRODUCTION

In many parallel computations, the workload needs to be periodically redistributed among the processors. On a distributed memory computer, this generally requires data structures associated with the computations to be transferred between processors. Many examples of this phenomena occur in scientific computing. When computational work varies over time, the tasks and attendant data must be redistributed to keep the workload balanced. Examples include: adaptive mesh refinement, particle simulations with short- or long-range forces, state-dependent physics models, and multi-physics or multi-phase simulations.

*This work was funded by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research and performed at Sandia, a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the U.S. DOE under contract number DE-AC-94AL85000.

A number of algorithms and software tools have been developed to repartition the work among processors (see, for example, [2, 5] 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 an asynchronous receive 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 simultaneously hold both the outgoing and the incoming data since incoming messages can arrive before 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 which can be performed. Second, the amount of memory required by this protocol is unpredictable, so setting aside a conservative amount of space is likely to be wasteful. And third, a general purpose tool for dynamic load balancing should be robust in the presence of limited memory. It was the construction of just such a tool which inspired our interest in this problem [4].

To address these problems, we propose a simple modification to the above scheme. 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. Since each phase can be expensive, it is important to limit the total 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 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 which is requested to be transferred from processor $i$ to processor $j$.

We now wish to perform the requested transfer in a sequence of phases. Let $t^l_{ij}$ denote the volume of data transfer from processor $i$ to processor $j$ in phase $l$, and let $A^l_i$ be the memory available to processor $i$ at the beginning of phase $l$. We will also use $R^l_i$ and $S^l_i$ to denote the total volume of data received and sent by processor $i$ in the phase $l$ (i.e., $R^l_i = \sum_{j=1}^{k} t^l_{ji}$ and $S^l_i = \sum_{j=1}^{k} t^l_{ij}$).

At each step the constraint of finite memory requires that $R^l_i \leq A^l_i$ for $i = 1, 2, \ldots k$. The available memory after each phase can be computed as $A^{l+1}_i = A^l_i + S^l_i - R^l_i$. Our objective is to find a schedule of transfers which obeys the memory constraint, and satisfies the transfer request in a minimal number of phases. We will call this the *minimum phase remapping problem.* Note that there is a corresponding decision problem: can a transfer request be completed in a specified number of phases?

In §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 the paper focuses on formulations and approximation algorithms which could be used in practice. In §3 we present a reduction of our problem to multi-commodity flow. We present a continuous relaxation of the problem in §4, and a practical approximation algorithm in §5.

Despite its practical importance, we are unaware of any previous work on efficient data transfers with limited memory. Some standard collective communication operations can be implemented in ways that limit memory usage, but the general problem we are proposing seems to be new. Cypher and Konstantinidou designed memory efficient message passing protocols [3]. However, their work addressed exchange of tokens as opposed to variable sized messages. And they didn't 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.

## 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. Recall that a Hamiltonian Circuit is a cycle in the graph that visits each vertex once. The directed Hamiltonian Circuit problem is known to be NP-Complete [6]. 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* which gets passed between processors, and possession of the token allows a processor to receive data in the next phase. In our construction, 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.

While the token is being passed in a cycle, the processors must not perform any other data transfers. But when the cycle is completed, they must be able to finish all their other communication operations. 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 which will percolate upwards with each phase, finally allowing all the data transfers to be completed. Given a Hamiltonian Circuit Graph $G = (V, E)$, we construct a data remapping problem with $\mathcal{P}$ as the set of processors and $T$ as the set of transfer requests as follows.

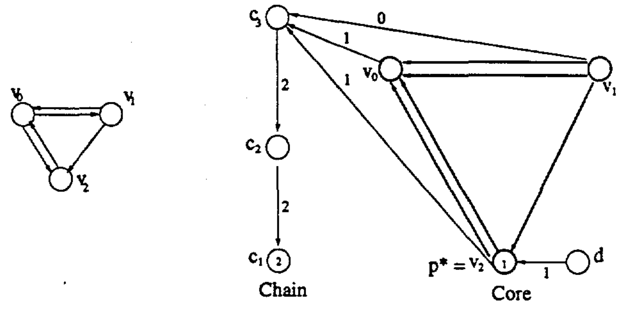

Figure 1: Construction for NP-Completeness proof.

- $\mathcal{P}$ has a processor $p_i$ for each vertex $v_i$ of $V$. We will refer to these processors as core processors. Each edge of $E$ is a unit-volume transfer in $T$.
- Add a chain of $|V|$ processors $\{c_1, \ldots, c_{|V|}\}$ to $\mathcal{P}$. Also, add transfer requests $(c_{i+1}, c_i)$ to $T$, 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 $p^*$. All other processors have no free memory.

Consider what happens as the data remapping occurs. In the first phase, $c_1$ will send its data to $c_2$, 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 single unit of free memory (the token) which started at $p^*$ will have meandered about, enabling some data to be transferred.

In phase $|V|$, processor $c_{|V|}$ has enough free memory to receive all of the data that needs to come to it 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 completion of phase $|V|$, each core processor $p_i$ graph has (indegree($p_i$) $-1$) units of free memory. (One processor might also have an additional unit of free memory from the token).

In phase $|V|+1$ core processor $p_i$ can now receive all the data that needs to come to it, minus 1. The complete set of 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. If there is a processor that was not visited by the token in phases 1 to $|V|$, then that processor cannot receive all its data in $|V|+1$ phases. 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 during phase $|V|+1$.

This argument leads to the following result.

THEOREM 2.1. *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.

The total amount of data that needs to be transferred is $|V|(|E|-|V|)+|E|+1$. The first term comes from the data being sent to the chain and within the chain. The second term reflects that data being redistributed within the core and the last term is the transfer from $d$ to $p^*$. This quantity equals $(|E|-|V|+1)(|V|+1)$. Since there are only $(|E|-|V|+1)$ units of free memory, the transfers can complete in $(|V|+1)$ phases only if all the free memory is used at every phase. So the transfers must proceed as discussed above.

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.

Notice that 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, so the problem is in NP. □

## 3. MULTI-COMMODITY FLOW FORMULATION

In this section, we present a multi-commodity flow (MCF) formulation to determine whether a given transfer can complete in a specified number of phases [1]. 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 optimally solve the minimum phase data remapping problem. This might be helpful for three reasons. First, some MCF problems can be solved relatively fast, 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.

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 want to 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 which occurs in the corresponding phase.

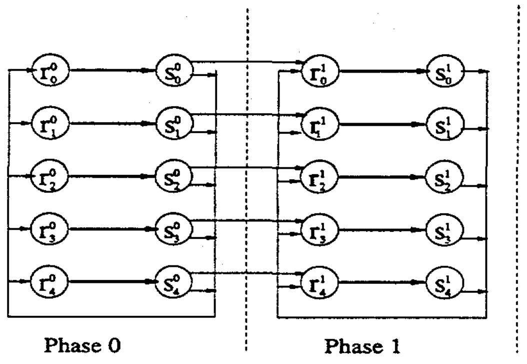

Figure 2: MCF graph for 5 processors and 2 phases.

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 which corresponds to data that will 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 available space to allocate memory for the incoming data before releasing the space for the data being shipped out, thus 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 $l = 1, \ldots, L-$

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 3.1. *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 $G$. So the feasibility of one solution implies the feasibility of the other. $\square$

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.

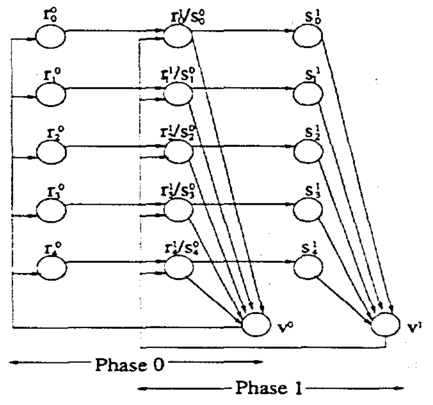

Figure 3: MCF graph after reduction.

## 4. CONTINUOUS RELAXATION

Although the multi-commodity flow formulation from §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 in 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 $L = \lceil \frac{T}{M} \rceil$ be the 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, we will use preprocessing and postprocessing phases to enable feasibility of the phases in between.

LEMMA 4.1. *If the following conditions are satisfied, the continuous version of the remapping problem can be completed in $L = \lceil \frac{T}{M} \rceil$ phases.*

*(i)* $S_i = R_i$ *for all processors.*

*(ii)* $A_i^0 \geq \frac{R_i}{L}$.

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. $\square$

LEMMA 4.2. *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 *(i)* and *(ii)* from Lemma 4.1, 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 $R_j - S_j > A_j^0$. So this rearrangement can be completed in a single phase.

Next, as a second part of the preprocessing step, processors $i$ with $A_i < \frac{R_i}{L}$ will transfer some of their outgoing data to processors $j$ with $A_i > \frac{R_i}{L}$. To avoid disturbing the first property, sending processors will also pass equal amounts of receiving assignment. Once 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 need to correct for the transfer of receiving assignments. This 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, so the postprocessing can be completed in one phase. $\square$

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 $R_j > S_j$. 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 4.3. *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 4.2, $\mathcal{R}$ can be completed by pre- and postprocessing steps, along with a transfer request $\mathcal{R}'$ satisfying conditions of Lemma 4.1. Notice that the total volume of data to be transferred $T'$ in $\mathcal{R}'$ is no greater than $T$ of $\mathcal{R}$, and the total available memory in the system does not change: $M = M'$. So by Lemma 4.1, $\mathcal{R}'$ 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. $\square$

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

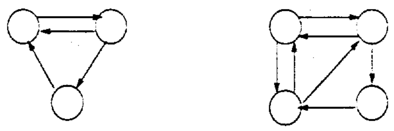

Figure 4: Catastrophic instance for continuous relaxation.

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 4.2, this available memory will be divided into two groups of processors, but the fractional transfers which 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.

## 5. EFFICIENT APPROXIMATION ALGORITHMS

In this section, we describe the basics of a family of efficient algorithms that provides 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 $M$ doesn't change between phases.

LEMMA 5.1. *The minimum number of phases in a solution is $\lceil \frac{T}{M} \rceil$.*

This bound can only be achieved if available memory is used to receive messages at each phase. So 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 it can't use, which frees up memory on processors which can use it. We will only park data that needs to be transferred eventually.

### 5.1 Parking

Parking aims to utilize memory that would otherwise be wasted. Consider a processor that received all its data and still has available memory. This memory cannot be utilized in subsequent phases, decreasing the total memory which is 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 is greater than available memory, and the quantity of this parkable data is $\max(0, k - m)$ units. The parkable data consists of data that eventually needs to be

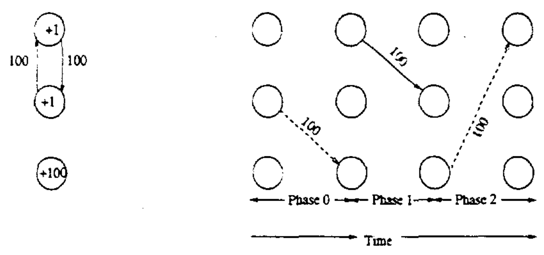

Figure 5: Example of the utility of parking.

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, 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 parking space.

LEMMA 5.2. *It is sufficient to park data at most once to get an optimal solution.*

PROOF. Assume there exists 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 to $p_1$, then we can rearrange the data movement as $D$ staying in $p_1$, and $p_i$ parking to $p_2$, 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 5.1.

- *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 5.3. *The total volume of data transferred by Algorithm 5.1 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 5.2, thus parked data is moved twice, and the total volume of data moved is $2T_p + T_d = T + T_p$. Because each parked unit of data enables at least one direct transfer, the algorithm guarantees that $T_p \le T_d$, Thus at most half of $T$ can be transferred through parking, i.e., $T_p \le \frac{T}{2}$, and the total volume of data moved is $T + T_p \le T + \frac{T}{2} = \frac{3T}{2}$. ☐

THEOREM 5.4. *Algorithm 5.1 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 5.3 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: $1 < x \le 2M$. From Lemma 5.3 we know that

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

But simple algebra reveals that

$$l - 1 \le \lceil \frac{(l-2)M + x}{M} \rceil$$

Combining these inequalities

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

and the result follows. $\square$

Combined with Lemma 5.1, Theorem 5.4 shows that Algorithm 5.1 is a 3/2 approximation algorithm for the minimum phase remapping problem. Without a tighter lower bound, this value of 3/2 is tight as illustrated by the example in Fig. 6.

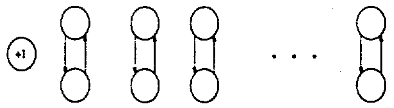

Figure 6: Example to show the tightness of the 1.5 bound.

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 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 to first 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. CONCLUSION

We studied the problem of moving large amounts of data among processors under memory constraints, which is required for applications where workload and associated data are periodically redistributed among processors. The problem arises when processors do not have enough memory to allocate space for their incoming data, before releasing the space for the 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 a 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 multi-commodity flow was presented. We showed how a continuous relaxation of the problem admits a simple solution with two more phases than that of the optimal solution, but it may be difficult to get a good discrete solution from this continuous one. Finally, we devised a practical approximation algorithm with a bound of 1.5 times the optimal solution.

We are currently implementing several of these approaches for use in the Zoltan dynamic load balancing tool [4]. We will report on our empirical comparisons in due course.

## 7. REFERENCES

[1] AHUJA, R. K., MAGNANTI, R. L., AND ORLIN, J. B. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] CYBENKO, G. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.* 7 (1989), 279–301.

[3] CYPHER, R., AND KONSTANTINIDOU, S. Bounds on the efficiency of message-passing protocols for parallel computers. *SIAM J. Comput.* *25*, 5 (1996), 1082–1104.

[4] DEVINE, K. D., HENDRICKSON, B., BOMAN, E. G., ST.JOHN, M. M., AND VAUGHAN, C. Zoltan: A dynamic load-balancing library for parallel applications – user's guide. Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

[5] HENDRICKSON, B., AND DEVINE, K. Dynamic load balancing in computational mechanics. *Comp. Meth. Appl. Mech. Eng.* (2000). Invited paper. To appear.

[6] KARP, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. Plenum Press, New York, NY, 1972, pp. 85–103.