Airline Crew Pairing Generation in Parallel

Diego Klabjan *
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0205
Tel: (404) 894-2366
Email:diego@isye.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Tel: (404) 894-2589
Email:schwan@cc.gatech.edu

June 8, 1999

Abstract

Crew pairing generation is a time consuming part of the complex process of airline crew scheduling. We present a parallel algorithm for this problem's solution and several implementations of this algorithm on a parallel machine. Linear speedup is attained by exploitation of the combined shared/distributed memory architecture of the underlying cluster computing engine, by development of a novel communication mechanism, and by creation of application-specific load balancing algorithms. Specifically, we distribute pairing generation across both the distributed nodes of the cluster machine and the multiple processors within each cluster node. The resulting parallel computation utilizes both processes mapped to nodes and within each process, multiple threads running on the nodes' multiple processors. Since this distribution should be done in accordance with the actual workload experienced on nodes, it is performed and re-performed repeatedly throughout the execution of the parallel program, using an application-specific

*This research is supported, in part, by NFS Grant DMI-9700285 to the Georgia Institute of Technology

dynamic load balancing algorithm. In addition, since the parallel program employs both threads and processes, we develop an application-specific communication mechanism, called active interchange, able to take advantage of threads to speed up load balancing and communication actions. Experimental evaluations of our approach utilize large-scale compute clusters using multiprocessor engines based on PC chips, including Pentium Pro and Pentium II processors.

Subject classifications: Transportation, scheduling, personnel; Computers/computer science.

Airline crew scheduling is the problem of assigning crews to flights. The problem has received considerable attention in recent years due to the high crew cost incurred by airlines (see e.g. Anbil, Johnson and Tanga [2], Anbil et al. [1], Hoffman and Padberg [19], Chu, Gelman and Johnson [6], Graves et al. [16]). The computational solution of crew scheduling involves solving a large set partitioning problem, with each member of the set representing a flight in a given schedule. The partitioned set defines the crew pairings being sought, where each pairing consists of a small sequence of working days, called duties. Typical pairings are comprised of 2-4 duties, and the number of subsets, i.e. pairings, ranges from 2 million for small problems, to 20 million, to 50 million and above for large problems.

Crew scheduling must be performed monthly. Current solutions implemented on mainframe machines typically require days of CPU time to compute acceptable schedules. Of this time, hours are consumed by pairing generation (see e.g. Graves et al. [16], Chu, Gelman and Johnson [6]). Therefore, speeding up the pairing generation process is a necessary step toward reducing crew scheduling time. Alternatively, by reducing pairing generation time, the quality of crew scheduling may be improved, by making additional time available for solving other parts of the complex methods used for crew scheduling, including solving the LP relaxation, finding integer solutions, and local improvement procedures. Finally, pairing generation is also a necessary element of crew recovery procedures Letovsky [23]. These procedures reschedule crews in response to schedule disruptions caused by bad weather, unforeseen gate delays, etc. In contrast to crew scheduling, due to the relatively small number of pairings, the pairing generation time in a recovery procedure could be run in seconds of time with the parallel approach described in this paper.

Our approach to parallelizing pairing generation relies on dynamic domain decomposition, where generated duties are distributed across multiple

processes running on a parallel machine. This permits the parallel application of the complex rules on which pairing generation is based. However, an issue with dynamic domain decomposition is the balance of workloads among processes. In our application, imbalanced workloads occur because some processes may have completed computing the pairings from their duties whereas other processes' computations are still in progress. Such imbalances strongly affect program speedup, particularly for smaller problems and during the final stages of a larger problem's solution when the number of duties remaining to be considered is relatively small, termed *tail effects* in the remainder of this paper. Toward this end, we develop and evaluate a novel mechanism for sharing duties and in general, computational tasks, among cooperating processes. We term this mechanism an *active interchange*; its implementation relies on the combined use of threads, messages, and processes in the cluster computing environment in which our parallel implementations are performed. Furthermore, we also study the impact of tail effects on the performance of parallel solutions to dynamic problems and suggest some approaches to addressing them.

The parallel machine used in this research is a cluster computing engine, in which standard PCs are networked via commercially available interconnects, thereby offering factors of cost/performance an order of magnitude superior to that of current parallel supercomputers or server engines. The PCs used in our work are 200 MHz Quad Pentium Pros as well as 300 MHz dual Pentium IIs, all of which are linked via point-to-point switched Fast Ethernet utilizing two Cisco 5500 network switches. The resulting heterogeneous nature of the computing engines (Pentium Pros and IIs) and of the underlying communication medium (within each Pentium Pros or II node, vs. across two such nodes) is typical of modern parallel computing environment. This heterogeneity and its relatively 'slow' internode communications are the two principal factors contributing to the importance of the workload balancing and active interchange mechanisms developed in this research.

The contributions of this paper are:

- a novel method of program parallelization utilizing both process- and threads-based parallelism and thereby effectively exploiting the heterogeneous nature of the underlying cluster compute engines;
- the development of an active interchange mechanism permitting the efficient implementation of the workload balancing schemes developed in this paper;
- workload balancing schemes that utilize randomization to ensure a fair

distribution of work items across distributed machines;

- a demonstration of high program performance, resulting in our ability to solve problems of sizes not easily solved on current mainframe machines; and
- a detailed study of tail effects due to workload imbalances in applications like ours and the development of approaches that effectively address such effects.

1 Airline Crew Scheduling: Background and Outline

1.1 Crew Scheduling

We start by stating some basic airline terminology. Consider a flight schedule of a given fleet. It consists of several flights, also called *legs*. Legs connect stations, where some stations are chosen as *crew bases*. A *duty* is a working day for a crew. Several duties are composed to form a *pairing*, which is a sequence of duties for a single crew. A pairing must start and end at the same crew base; it typically consists of two to four duties. A pairing is subject to many FAA rules, including a maximum allowed flying time in a duty, a bound on the elapsed time of a duty, a bound on the minimum overnight rest, etc. In addition, there are union rules, concerning the maximum number of duties in a pairing, the minimum cost of a pairing, the maximum time away from base, and others.

A *crew schedule* is composed of a set of pairings so that each leg is in exactly one pairing and is repeated to cover the time horizon, typically one month. The airline crew scheduling problem is trying to find a subset of pairings that partition all of the flights in the schedule. The crew scheduling problem is modeled as a set partitioning problem, with its columns representing pairings. The pairing cost is nonlinear; it typically consists of a maximum of 3 quantities. The nonlinearity of this cost, the huge number of pairings, and the complex FAA and union rules are the principal reason for the difficult nature of the problem. For example, we estimated that a typical fleet of 200 flights has a billion pairings.

Pairing generation is also used in solutions to the crew recovery problem. Namely, during the recovery process, new pairings have to be generated in real time in order to deal with disruptions in operations. The difficulty of this problem is not the huge number of pairings to be considered but the need

to generate them as fast as possible. A parallel implementation of pairing generation can meet this need. Some details on crew recovery are explained in Letovsky [23], where the biggest problem being solved involves 15000 pairings. This number would increase considerably for large disruptions like thunderstorms at major airline hubs (i.e., crew bases) for which pairings have to considered in which crews and flights cannot get back on schedule on the same day.

1.2 Methods of Parallelization

Dynamic domain decomposition has been used extensively for the parallel implementation of branch and bound optimization methods Blake, Bo and Schwan [3], Gedron and Crainic [15], Eckstein [11, 12]. In comparison, in our problem, the size of each individual duty shared among cooperating processes is relatively small, thereby implying the need for low latency rather than high bandwidth in interprocess communication.

The load balancing algorithm described in this paper is fully distributed, initiated by the receiver, and scales well for the number of processors used in our work (i.e., hundreds of processors). For larger target machines, we also propose a semi-distributed version. Most of the load balancing algorithms in the literature attempt to balance machine loads when those machines are used by multiple end users Hac and Jin [17], Heederik et al. [18], Reeves and Willebeek-Lemair [29]. These algorithms share with our work the notions of (1) trying to determine the profitability of moving substantive portions of a machine's load rather than moving load one job at a time, and (2) attempting to estimate current and future load imbalances before load is migrated. They differ from our work in their description of 'load' as the number of jobs to be migrated vs. the application-specific notions of load considered in this paper. A general method for accommodating such application-specific load formulations is to insert user-defined subroutines into load balancing subsystems, as described in Reeves and Willebeek-Lemair [29].

'Active messages' Culler et al. [8], Culler and Lumetta [9], Bo and Schwan [5], Mainwaring and Tucker [24] is an asynchronous communication mechanism that is generally useful for applications like ours, in which communication and computation should overlap in an unpredictable manner Blake and Schwan [4]. The implementations of active messages described in Culler et al. [8], Culler and Lumetta [9], Bo and Schwan [5] rely on interrupts to attain rapid responses to incoming communications. In comparison, the active interchange mechanism we have developed utilizes additional user-level threads and application-specific means of polling on incoming communica-

tion channels. It is unclear whether its replacement by active messages would result in additional performance benefits.

There has been research in Computer Science that is trying to create more powerful communication/computation libraries; it includes Nexus Foster, Kesselman and Tuecke [14] for wide area networks and Chant for threads-based message communications Cronk, Haines and Mehrotra [7]. We do not employ these libraries in order to create optimization codes portable to the wide variety of parallel computing platforms now in common use by end users solving crew scheduling problems.

We are not aware of any previous work on parallel methods for airline crew pairing generation. Additional information about sequential methods for pairing generation may be found in Letovsky [23] and Hu [20].

1.3 Overview of Paper

In Section 2, the parallel algorithm is described in detail, and we state several conceptual issues concerning this method of parallelization along with proposals for addressing these issues. Section 3 outlines the algorithm's implementation on the cluster machines employed in this work. It describes implementation issues arising on these platforms, and proposed solutions. Section 3 also outlines our computational experiments and explains the architecture and software used. Results are summarized in the last section.

2 The Crew Pairing Algorithm

2.1 Approach

There are two possible approaches to performing pairing generation sequentially: based on the flight time network or based on the duty timeline network. The flight time network has an arc for each leg in the schedule and also has arcs connecting legs. The duty timeline network has an arc for each possible duty and arcs for possible overnight duty connections. The latter first generates all of the duties and then generates pairings from duties, the former generates pairings directly from flights.

We choose the duty-based approach since it is faster and more suitable for parallelization. Suitability for parallelization derives from the number of flights in the flight-based approach, which ranges in hundreds, vs. the number of duties, which ranges from 10,000 to 200,000. One potential drawback of the duty-based approach rendering it difficult to use in the past is its large memory requirement, since the duty timeline network must be stored

in main memory. Storage needs for the duty- vs. flight-based approach are somewhat ameliorated by the fact that the required storage per pairing in the duty-bases approach is significantly smaller than with the flight-based approach. As a result, storage problems do not exist even for the largest problems (i.e., for the biggest fleets) made available to us for our work by a major U.S. airline. For reference, the machines we use have 256Mbytes of main memory per processor. If less memory is available, additional reductions in memory needs may be obtained by using compact storage schemes like those presented in Hu [20].

While this paper uses duty-based pairing generation, its method for pairing generation may also be applied to flight-based networks, by taking the following steps. First, all duties are generated with the origin station residing at a crew base. This does not result in large memory needs as the number of such duties is much smaller than the total number of duties used, typically less than 10% of this number. Next, the pairing generation procedure is initiated using these duties and then extends them to pairings using the segment timeline network.

2.2 Algorithm Description

Let CB be the set of stations that are crew bases. The start of a duty is the departure station of the first leg in the duty. There is a natural order on duties that start at the same station. This order is based on the departure time of the first leg in the duty. We assume that duties are already ordered in this way (if one generates duties from the flight time network using the depth-first search, then the duties are generated according to the above order). Let D_{cb} be the set of all duties that start at the station cb. We call a duty a starting duty if it starts at a crew base. We can generate all of the pairings starting at a given duty by performing a depth-first search and perform pruning whenever a pairing violation occurs. Assume that a procedure denoted by DFS(d) enumerates all of the pairings that have as a starting duty a given duty d.

The sequential algorithm may be stated as follows:

Algorithm 1 Sequential Algorithm (SA)

```
1: for all crew basis cb in CB do
2:    for all duties d in D_cb do
3:       DFS(d)
4:    end for
5: end for
```

The sequential algorithm is parallelized by performing DFS in parallel. With this approach, a load balancing problem arises from the fact that the computation time of DFS(d) for a given duty d cannot be estimated statically. This prompts us to use a simple, initial distribution of duties across processors, followed by a dynamic load balancing strategy. The initial distribution used is one that uniformly and at random distributes starting duties to processes.

Dynamic load balancing is necessary because all of the computational experiments we have performed indicate a high variance in computation times for a random set of duties. Load balancing is initiated by receivers of load. Specifically, whenever a processor becomes idle, it queries all other processors about the numbers of starting duties they have not yet processed. It then chooses the process i with the highest such number. The process i then sends half of its remaining starting duties to the idle process. This method of load balancing aims to create equal workloads and therefore, equal execution times for all of the processes sharing loads.

The parallel algorithm is described in Algorithm 2 (assuming there are p processes).

Algorithm 2 Parallel Algorithm (PA)

1: **for all** crew basis cb in CB **do**
2: **for all** duties d in D_{cb} **do**
3: Send the duty d to a randomly chosen process.
4: **end for**
5: **end for**
6: For $i = 1, \ldots, p$ let D_i be the set of all starting duties at the process i.
7: **for all** $i = 1, \ldots, p$ **in parallel do**
8: **while** $D_i \neq \emptyset$ **do**
9: **while** $D_i \neq \emptyset$ **do**
10: Remove a duty d from D_i.
11: DFS(d)
12: **end while**
13: $l = \max_{j=1,\ldots,p}\{|D_j|\}$
14: Let the processor $j = \text{argmax}_{k=1,\ldots,p}\{|D_k|\}$ send half of the starting duties in D_j to the current process i. Append the received duties to D_i.
15: **end while**
16: **end for**

A major iteration of the algorithm is comprised of Steps 8-15.

Algorithm 3 Modified Parallel Algorithm (MPA)

1: **for all** $i = 1, \ldots, p$ **in parallel do**
2: $k = 1$
3: **while** $D_i \neq \emptyset$ **do**
4: **while** $D_i \neq \emptyset$ **do**
5: Remove a duty d from D_i.
6: DFS(d)
7: **end while**
8: $k := k + 1$ whenever the process i is involved in a query.
9: $s = 1, l = 0, u = b_k, S = \{1, \ldots, p\}$
10: **while** $S \neq \emptyset$ or $l < 1$ **do**
11: Randomly choose u processes from S and query them for the number of remaining starting duties.
12: Let l be the maximum of all the numbers that were received back.
13: Let j be the index of the processor where the maximum was attained.
14: Remove from S all the indices of processors that were queried.
15: $u := u \cdot a_s, s := s + 1$
16: **if** $l > 1$ **then**
17: Let the processor j send half of the starting duties in D_j to the current process i. Append the received duties to D_i.
18: **end if**
19: **end while**
20: **end while**
21: **end for**

2.3 Discussion

Two issues exist with the algorithm presented above. First, the load balancing method it employs will not scale to thousands of processors or to target distributed machines that are not as well-connected as the cluster engines used in our work. This fact is principally due to the request initiator's broadcast for work to all other processes and its subsequent desire to request work from the most loaded process. Such global operations are inherently non-scalable Blake and Schwan [4]. Second, as the parallel algorithm nears the 'end' of its execution, more and more processes will be asking for work, therefore generating a large number of queries and unsuccessful query responses. These actions constitute pure overhead as they do not advance the solution to the problem of pairing generation. We term this problem a *tail effect*, because its visible effect is a marked decrease in effective speedup toward the end of the problem's solution. The proposed algorithm enhancements described next address these two issues.

2.4 Algorithm Enhancements

The scalability of load balancing is improved by altering the algorithm outlined above as follows. Rather then querying all other processes for work as performed in lines 13 and 14 in (PA) above, additional work is requested from a subset of processes. This implies that work will not always be equally distributed since the remote process with maximum load may not be contained in the subset addressed by a query. Worse, the subset may not contain any process possessing work to be returned. These two problems are addressed by asking for additional work in several rounds. The first round queries a subset of processes. If successful, the parallel processing of duties proceeds, else a significantly larger number of remote processes is queried. Furthermore, to perform effective workload balancing toward the end of each program run, we increase the initial number of processes queried with increasing major iteration numbers. Stated precisely, let $a_1, a_2, a_3, \ldots$ be any increasing sequence of integers and let $b_1, b_2, b_3, \ldots$ be a nondecreasing sequence of integers. The first sequence determines the manner in which to increase the number of remote processes queried in response to an unsuccessful query. The second sequence states the initial number of remote processes to be queried, where the sequence iterator corresponds to the number of major iterations performed on all processes by parallel pairing generation. Suitable methods for choosing both sequences are described below.

The second problem identified above, termed 'tail effect', may be ad-

dressed by suitable selection of the values for a_i and b_i. First, to address the decreasing probability of finding work with increasing major iteration numbers and when querying only a fixed size subset of remote processes, we choose some step function with a fixed step interval and step size of 1 for b_i. Second, for a_i, we use sequence values of $a_i = 2^i$, which implies that a rapidly increasing number of remote processes is queried upon an unsuccessful query, thereby reducing the total number of remote queried issued.

An alternative solution to the use of fixed sequences like $a_i = 2^i$ in the selection of values for a_i and b_i is one that performs selection in response to the actual execution times of major iterations. The idea is to choose sequence values adaptively, in accordance with the actual load experienced during each iteration. A small execution time implies a small amount of work available locally and received last time a query was issued, whereas a large execution time implies that plenty of work remains to be completed by the parallel program.

For the statically identified sequences and the adaptive choice of sequence values, the parallel program implementing both, called MPA, may be derived from the algorithm (PA), by replacing steps 7-16 with Algorithm 3 on page 9.

3 Computational Experiments

3.1 The Parallel Computing Environment

All computational experiments are performed on clusters of machines. Two clusters are used, the first consisting of 16 200MHz Quad Pentium Pros and the second comprised of 48 300MHz Dual Pentium IIs, resulting in 160 processors total available for parallel program execution. All machines are linked via 100 MB point-to-point Fast Ethernet switched via a Cisco 5500 network switch.

Using the terminology from Quinn [28], it is worth noting that this parallel computing platform is a multicomputer where each node is a uniform memory access multiprocessor (UMA). 16 nodes have 4 processors each, and 48 nodes have 2 processors each, where the processors within each node are linked via shared memory. This results in two hardware distances existing between processors, within each node via shared memory and between two nodes via switched Ethernet. Furthermore, for cross-node communications, congestion problems may occur due to the shared use of the fast Ethernet by the multiple processors on each node. Finally, note the difference in CPU speeds between the Pentium Pro vs. Pentium II nodes. This must be taken

into account for load balancing.

The operating system used is Sun Solaris x86 Kleiman et al. [22], Eykholt et al. [13], Version 2.5.1, which offers facilities for parallel computing like 'remote shell' (rsh commands), global file system support via NFS, and parallel computing libraries like MPI or PVM.

In summary, the cluster machine we use is representative of typical machines of this type, in its relatively slow internode communications and its good cost/performance ratio vs. specialized parallel machines like the CM-5, the Intel Paragon, or the IBM SP-2 machines. The Intel cluster machine also shares aspects with modern parallel machines like the IBM SP-3, in its use of multiprocessor nodes, with 8 processors/node used in the IBM SP-3 vs. the 4 processors used in our Intel cluster. The new Intel cluster available to us in 1999 will also use 8 processors/node, by utilizing the new 8-way Xeon boards.

3.2 Implementation Basis

The parallel implementations of pairing generation (MPA) use the MPI message passing interface, Message Passing Interface Forum [25], MPICH implementation Version 1.0, developed at Argonne National Labs. The MPI message passing standard is widely used in the parallel computing community. It offers facilities for creating parallel programs to run across cluster machines and for exchanging information between different processes using message passing procedures like broadcast, send, receive and others. However, Version 1.0. of MPICH on Solaris does not exploit information about different distances between machines, as done by MPICH implementations on SGI Origin multiprocessors that use shared memory to exchange messages between two processes residing on the same multiprocessor node. In other words, with MPICH 1.0, multiple processes on the same node communicate via MPI messages.

As with MPI, a portable threads standard is used in MPA's implementation: Pthreads (POSIX threads) 4, Nickols, Buttlar and Farrel [27]. The Solaris Pthreads library implements POSIX threads by mapping them to kernel light-weight-processes (LWP) Kleiman et al. [22], Eykholt et al. [13], with a unique kernel thread allocated to each LWP. As a result, it is straightforward to 'bind' threads to processors. This is important to our work as it eliminates unnecessary context switches between multiple threads running on the same multiprocessor node, as long as the number of threads does not exceed the number of available processors on each multicomputer node.

3.3 Parallel Implementation of Pairing Generation

3.3.1 The Active Interchange

Our first implementation was exclusively a message passing one. Results obtained using only MPI were poor and are not reported in this paper. One reason for these poor results is the lack of active messages in MPI. Namely, when a process becomes idle, it has to wait to get replies from other processes that are working on their own duties. Since it can take a long time for DFS to finish, an idle process can wait for new work for long time. This is unsatisfactory, resulting in no speedup above 40 processes even for large problem sizes.

In response to the need for activity, we have developed a mechanism termed an *active interchange*. The basic idea is to allow a process to be interrupted at any time while in the DFS phase, serve the request received, and then resume the DFS phase.

The implementation of active interchanges uses threads, with each process possessing a minimum of two threads. One thread, the working thread, does the actually enumeration, i.e. the iterations 9-12 in (PA). The other thread, the controlling thread, takes care of all message passing calls. Whenever there is a request for sending the number of remaining duties to an idle process, the working thread is suspended and the controlling thread takes over. When it finishes, the working thread resumes the DFS procedure. In order for such context switching to be performed as described above, the controlling thread has to have higher scheduling priority than the working thread. For the active interchange to function properly, the thread scheduler must suspend the lower priority in favor of the higher priority thread, then resume the lower priority thread when the higher priority thread blocks on message calls.

3.3.2 Load Balancing

The protocol for load balancing is as follows. An idle process sends a message to every other process. Each target process returns the number of remaining starting duties it possesses. The idle process then chooses the target process i with the highest load and sends it a request for load. Process i shares half of its starting duties with the idle process. A race condition can occur when using the above protocol. Specifically, if there are two idle processes, they may both select the same target process i for load sharing. In that case, only one of these idle processes will succeed in receiving substantial load; the target process i and the second idle process will each end up with half

the loads of the successful idle process. We were not able to observe this race condition in practice and therefore, did not amend the load balancing protocol to address it.

3.3.3 Implementation Issues

The current implementation of MPICH is not thread-safe, resulting in problems when multiple threads simultaneously use MPICH communications. In response, our current implementation places all MPI calls into the single controlling thread. This thread polls for messages and then processes them based on their tags.

There are two different hardware distances in our cluster computing environment. Our implementation utilizes this fact by using multiple worker threads within each node, which directly communicate via shared memory. Namely, if a node has i processors, then the process running on it creates i working threads and a single controlling thread. The working threads share the set of starting duties D_i.

Our processors have different CPU speeds, but each node's processors are homogeneous. Let p_i be the number of processors at node i and let cpu_i be the speed of processors at node i. We have evaluated cpu's beforehand by running small instances of the sequential algorithm (SA). In our computing environment, there are two different values for cpu's.

Let p be the number of nodes in the environment and let X be a random variable with the domain $1, \ldots, p$ and

$$\mathrm{P}\left[X = i\right] = \frac{cpu_i \cdot p_i}{\sum_{j=i}^{p} (cpu_j \cdot p_j)}$$

Then we replace step 3 of (PA) with the following step

3: Sample a random node based on the random variable X. Send the duty d to the sampled node.

We also need to replace steps 13-14 of (PA) to reflect the above observation. The modified steps read

13: $l = \max_{j=1,\ldots,p}\{\frac{|D_j|}{cpu_j \cdot p_j}\}$

14: Let the processor $j = \mathrm{argmax}_{k=1,\ldots,p}\{\frac{|D_k|}{cpu_k \cdot p_k}\}$ send half of the starting duties in D_j to the current process i. Append the received duties to D_i.

The same modifications need to be done for (MPA) algorithm.

4 Results

Due to the random distribution of starting duties, for each reported run we executed our code 3 times and took the average of the 3 execution times. The variance of a typical run is shown later.

4.1 Shared vs. Distributed Memory

We start by showing the difference between an implementation having one process per processor (pure MPI) and one that has one process per node (MPI with threads). This experiment is carried out on a medium size instance and on the cluster of 14 200 MHz Quad Pentiums. As we can see from Figure 1, the mixed MPI/threads implementation substantially outperforms the non-threaded code. These results are due to the fact that the use of threads reduces communication overhead. Performance differences are particularly pronounced for small numbers of processors. This is because the communication burden is relatively larger for these cases. Furthermore, the pure MPI code exhibits a clear 'tail effect', which is evident in Figure 1 from the lack of speedup experienced by this implementation for large numbers of processors. Measurements not shown in this figure determine this speedup degradation to be due to dropoffs in speedup toward the end of MPI program runs, where too many idle processors are requesting new work.

4.2 Speedup

The results depicted in Figures 2 and 3 demonstrate the good performance attained with the mixed MPI/threads implementation, when using the (PA) algorithm. These runs are performed with three data sets, using 5 crew bases and 97, 150, and 203 legs, respectively. One interesting insight from these measurements concerns the heterogeneous nature of modern cluster machines. Specifically, given that our environment consists of two types of nodes (200 and 300 MHz, respectively) configured as quad vs. dual shared memory multiprocessors each, it is actually not clear whether to employ more strongly connected, slower (i.e., quad Pentium Pro) nodes or more weakly connected, faster (i.e., dual Pentium II) nodes for program runs. The results depicted in Figure 2 show speedup when employing a mixed environment comprised of an equal number of 300 and 200 MHz nodes. The other figure depicts speedup when maximizing the number of 300 MHz nodes. It is apparent from these measurements that this application's com-

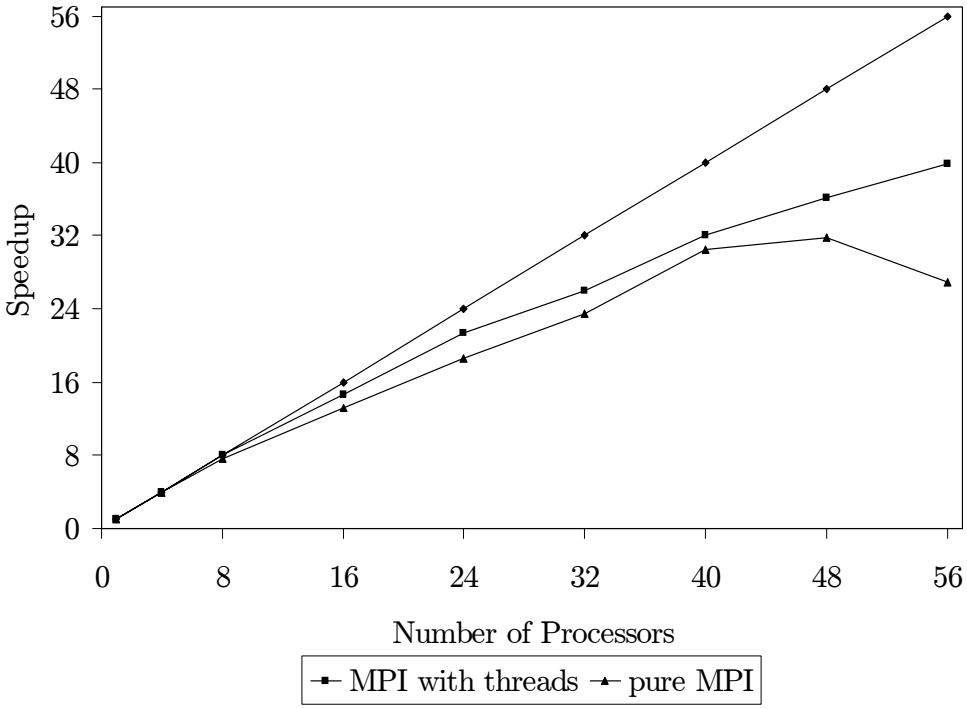


Figure 1: The comparison of pure MPI implementation and MPI with threads implementation

putation vs. communication ratio is such that faster processors should be employed whenever possible, as both speedup curves are close to linear, with diminishing returns for larger numbers of processors due to the limited problem sizes used in these experiments. At the same time, these measurements confirm our intuition that a 'better connected' machine results in improved speedup, as evident from the slightly better speedup in Figure 2 vs. Figure 3. However, while speedup is better with the well-connected machine, actual execution times are superior with the faster processors.

Another interesting observation from these results is that for smaller data sets, the ratio of communication to computation changes significantly, due to the decrease in the total amount of computation performed prior to the communications executed for purposes of workload balancing. As a result, for the smaller input data, actual execution times do not differ much when using 200 vs. 300 MHz nodes, with actual execution times being identical

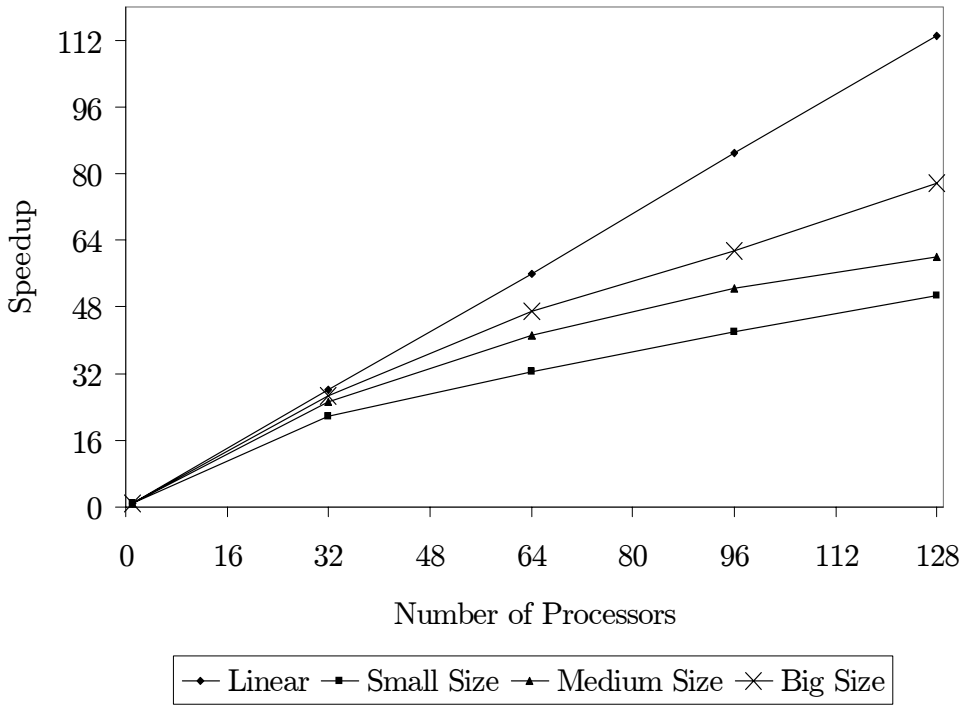


Figure 2: Half 300 MHz nodes, half 200 MHz nodes

for both cases when using 128 processors. It is clear that for smaller problem sizes, the time for DFS is short and hence the communication in applying the load balancing scheme has more weight. This implies that the choice of execution sites strongly depends on both the nature of the parallel program as well as the problem sizes it executes. We know of no automatic mapping methods that can deal with this problem prior to a program's execution.

The final insight from these experiments concerns limitations to total program speedup due to the aforementioned 'tail effect'. Essentially, toward the end of each program run, a limited number of problems remain to be solved, by a relatively large number of processors that are 'looking for work' in order to solve these problems. The controlling threads cannot be shut down until the generation is done on all nodes and hence there is communication congestion toward the end. We present one solution to this problem in Section 4.5 below. In addition, an inherent overhead (i.e., sequential component) in this program's parallel implementation is the required broadcast

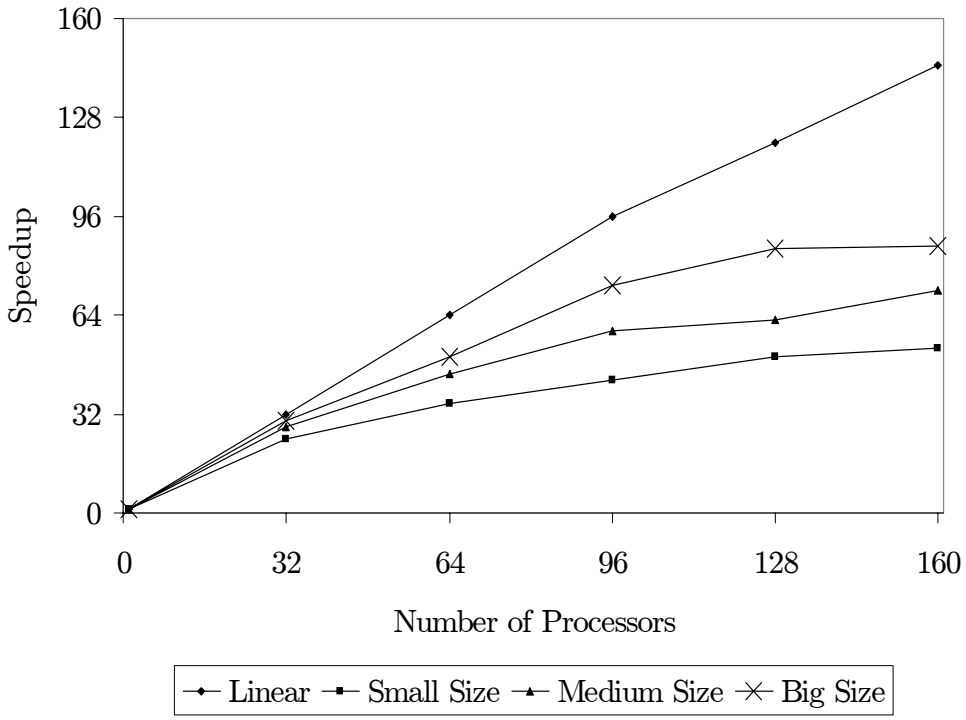


Figure 3: As many 300 MHz nodes as possible

of the duty timeline network.

4.3 Execution Time Variance

(PA) and (MPA) are randomized algorithms since starting duties are distributed randomly. However, we observed no significant variance in total program execution times. Even for our largest multi-hour runs, variance never exceeded 2 minutes. As expected, variance in execution times increases with problem size. Figure 4 shows a typical variance distribution for a run on 40 200 MHz Quad Pentium processors.

4.4 Hardware Effects

The performance of the (PA) algorithm depends on the underlying parallel computing environment, i.e. the computation/communication ratio supported by this environment. In order to demonstrate tradeoffs, we simulate

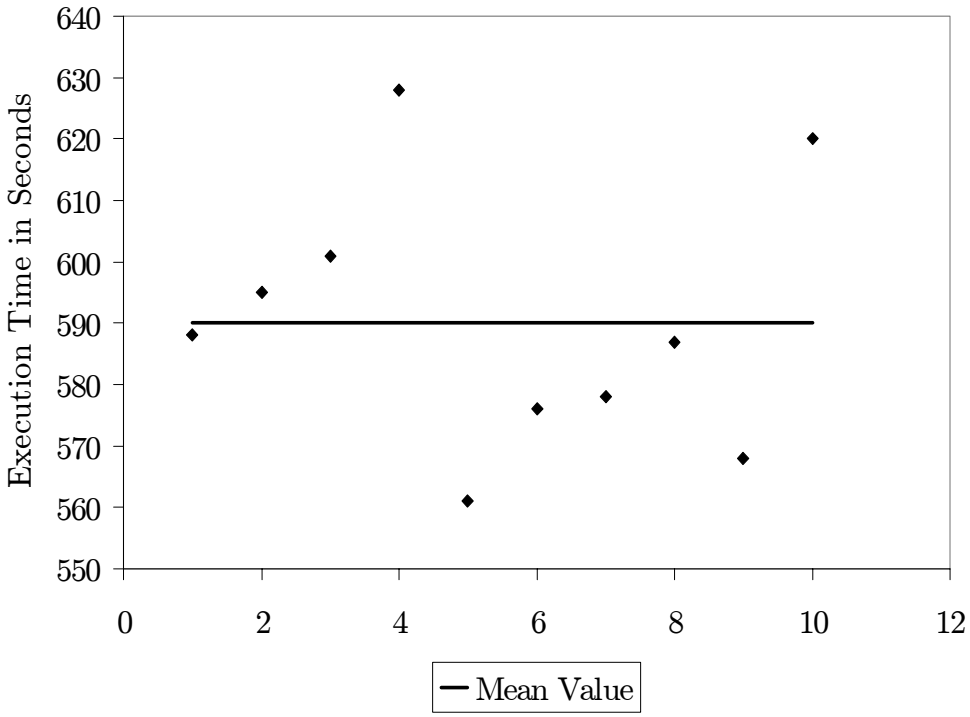


Figure 4: The variance

a more weakly connected environment by artificially increasing communication while not changing the total computation performed by the parallel program. Specifically, instead of sending a message of size t bytes, $50 \cdot t$ bytes messages are sent. Figure 5 demonstrates that there is a significant impact on execution time. This impact is primarily due to the increased communication load derived from the (PA) algorithm's broadcast of the duty timeline network. The load balancing scheme employed in this application is not bandwidth-limited, as most of its messages are quite small (a few bytes). Experiments in which only load balancing message sizes were increased did not result in slower program execution times.

In general, the experiments described above indicate that cluster machines like ours will not perform well for a broad class of parallel applications exhibiting computation to communication ratios that are less favorable than those in this application.

A more detailed study of hardware-related performance effects is one

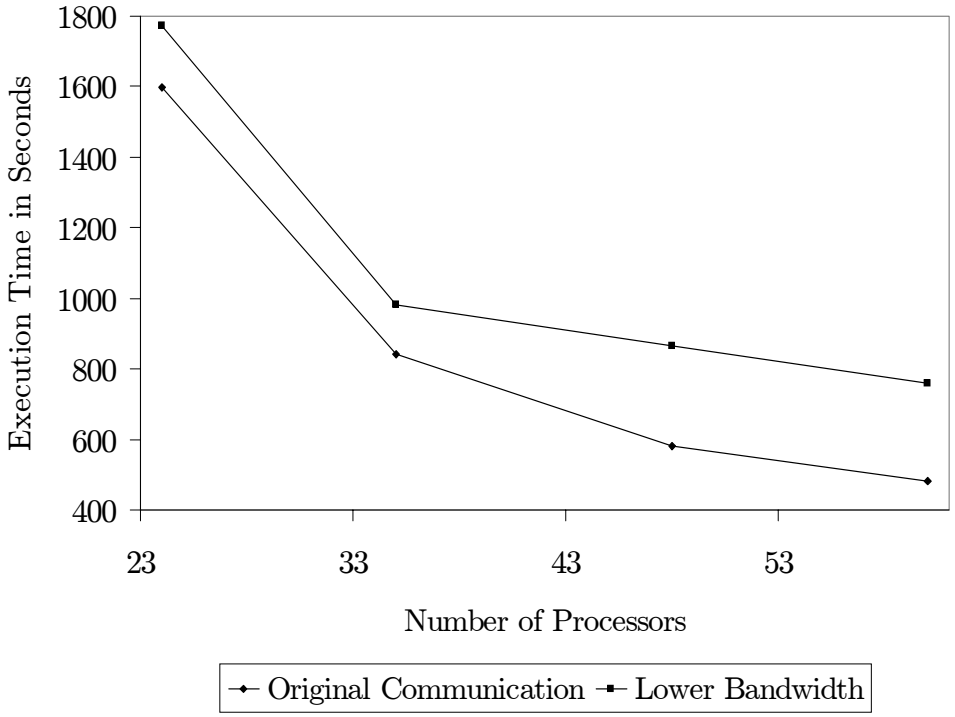
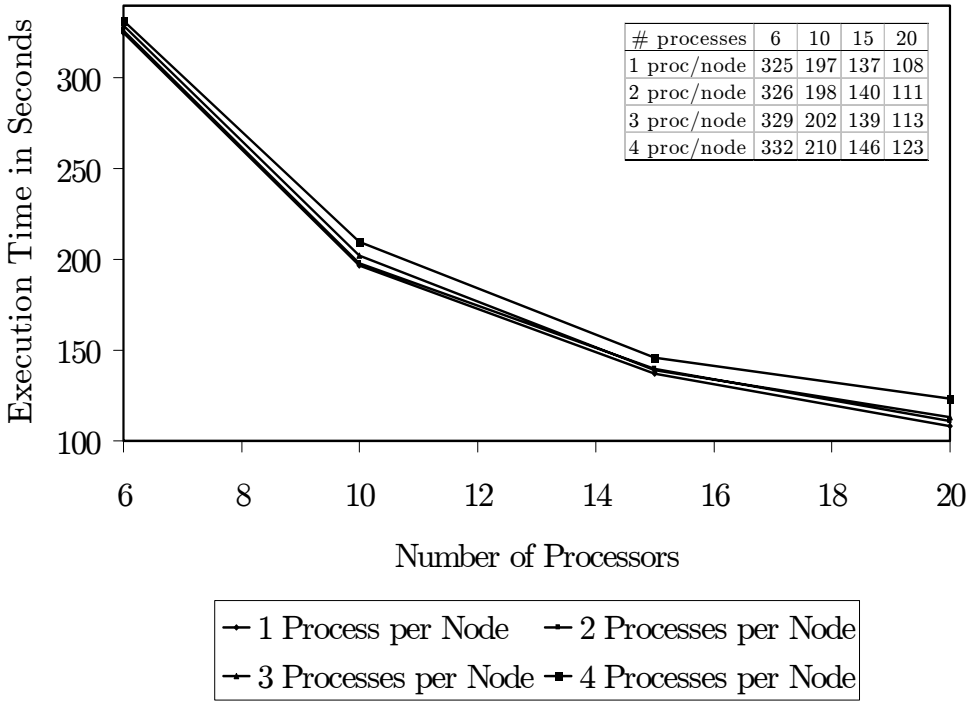


Figure 5: Effect of bandwidth

that considers the way in which each cluster node is networked. Specifically, in our machines, each node is connected to the network with one networking card, and a switch is used to enable concurrent communication across all nodes. While cross-node communications are concurrent, we claim that congestion will occur within individual nodes when its multiple processors simultaneously access its single network device. The measurements shown in Figure 6 confirm this claim. In these measurements, the pure MPI implementation is run on the cluster of 16 200 MHz Quad Pentium nodes. For a fixed number of processors, either 1,2,3 or 4 processes are placed onto each node. For example, in the first test, for 20 processors, 12 processes are placed onto 12 nodes and the remaining 8 processes onto 4 nodes. In the second test, 2 processes are placed onto 10 nodes. The third test uses 6 nodes with 3 processes per node and 1 node with 2 processes per node. The last test with 20 processors uses 4 processes per node and 5 nodes. In all of these tests, the same initial configuration of starting duties is used (i.e., we

use the same seed for the random number generator).



# processes	6	10	15	20
1 proc/node	325	197	137	108
2 proc/node	326	198	140	111
3 proc/node	329	202	139	113
4 proc/node	332	210	146	123

Figure 6: Networking card effect

Given the fact that all of the tests described above use the same number of processors, total execution times should be identical for all tests. This is not the case, as evident from Figure 6. Instead, performance steadily degrades with an increasing number of processors mapped to each node. We believe this degradation to be due to communication overheads experienced with the implementation of MPI communication primitives and the single network card resident on each nodes.

Two insights may be derived from these measurements. First, by using threads on multiprocessor nodes, communication overheads within each node may be reduced and therefore, performance may be increased significantly. Second, when multiprocessor nodes are networked, it may be important to employ multiple, not single network links for each node.

4.5 Tail Effect Study

Our experimentation with tail effects involves the (MPA) algorithm. Since the improvements promised by (MPA) concern scalability and communication speed, we use the pure MPI implementation with 16 200 MHz Quad Pentium nodes, thereby creating noticeable communication overheads. The difficulty in using (MPA) is the choice of sequences a and b. We observe that the loop 10 is rarely executed more than once. Hence the choice of the sequence a has little impact on total performance. We find that $a_i = 2^i$ works well for the problems considered in our work.

The choice of sequence b has significant impact. b_i is the expected number of processes that are queried at iteration i. The larger the iteration index, the harder it is to find work and hence b_i should be larger as well. If *init* is the initial value, then we choose $b_i = init + i$. *init* is the number of queries to be performed by the first idle process. *init* should be large if the number of starting duties is large, therefore depending on the size of the input data. Let n be the number of starting duties and let t be the wall time from the start of the algorithm until the first process becomes idle. We use the formula $init = F \cdot n/t^2$, where F is a constant that should depend on the target hardware. Note that this formula also indirectly captures the number of processes, since t depends on this number.

The first process that becomes idle records t and broadcast it to all other processes. The index i in the implementation is not the index of the major iteration because determining this number would require additional communication. Instead, i is the number of queries in which the given process has been involved up to the point of becoming idle.

F is chosen as follows. We use some fixed input data and a fixed, reasonably large number of processes. We determine the optimal *init* value for this fixed configuration by performing several runs each with different parameter value. From these runs, we determine the parameter resulting in the best execution time. We then compare the computed value with the formula and derive the appropriate value for F.

As the algorithm is approaching the end of its problem solution, it becomes difficult to find new work and hence, substantial time is spent in applying the load balancing scheme. We have termed this problem the 'tail effect' in dynamic load balancing.

To address this tail effect, the load balancing procedure used in our work is changed as follows. When the maximum number of duties on the queried processes falls below some given number, called the *tail number* and denoted *tail*, then the process stops applying the load balancing procedure. The

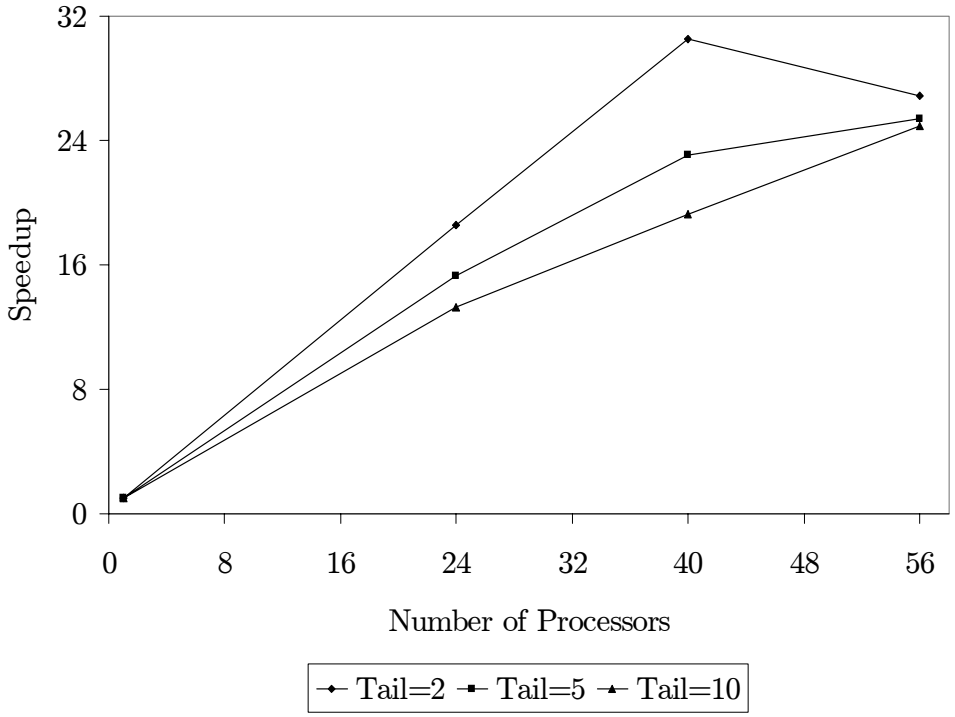


Figure 7: Tail effect

default value used in results presented so far is $tail = 2$, namely, we apply the load balancing steps until the very end. As can be seen from Figure 7, the tail number should be small if the number of processes in relation to problem size is small, since in these cases, the ratio of total time spent in solving a problem vs. time spent in load balancing is favorable. When this ratio is unfavorable (e.g., smaller problem size and a larger number of processes), then a larger tail number is desirable for two reasons: (1) the effect of stopping load balancing on total execution time is small and (2) the processors that have become idle due to their maximum loads falling below the tail number may be used by other application programs, thereby increasing machine utilization.

The discussion in the previous paragraph suggests that there is no single fixed value for tail number that is suitable for all problem and machine sizes. To address this issue, rather then choosing a fixed value of *tail* and shutting down all processes receiving new loads below this value, we apply

the following heuristics. First, if the maximum number of duties on the queried processes falls below the tail number, then these processes are shut down with probability tp. As a result, there will only be few processes that survive this shutdown procedure and continue to receive work from other survivors. Second, for determining suitable tail numbers, we use a strategy for parameter tuning similar to the one employed for the *init* parameter. Namely, let td be the average execution time of the DFS function. We choose $tail = F_1/td$. For the tail probability tp we use the formula $tp = F_2 \cdot p/(F_3 + td)$, where F_2 and F_3 are constants and p is the number of processes. We compute the constants in these formulas by computing several optimal probabilities and then using regression analysis to determine the 'best fit' values.

A summary of formulas appears below:

$$b_i = i + \frac{F \cdot n}{t^2} \qquad tail = \frac{F_1}{td} \qquad tp = \frac{F_2 \cdot p}{F_3 + td}$$

The parameter values for our hardware are $F = 3.5, F1 = 0.96, F2 = 0.02$, and $F3 = 1.7$.

The results are presented in the table below. The second column corresponds to the (PA) algorithm, whereas the third column reports the execution times in seconds for the (MPA) algorithm, with the choice of parameters described above. The improvements are not large, but we suspect that they will be more significant for problems with worse computation/communication ratios than the ones used in our work.

	(PA)	(MPA)
$p = 64$	570	497
	482	480
	535	500
	540	490
	1245	1145
$p = 50$	570	500

The algorithmic enhancements presented in (MPA) can be used in other applications. In order for this fact to be true, the domain being decomposed must be irregular, which implies that the execution times on decomposed domains have high variance. In addition, these execution times must be relatively short compared to communication overheads (i.e., program granularity must be small). There are no advantages in using these enhancements if processes run for a long time before becoming idle, i.e. if communication

times are small in comparison with process execution times. We believe the study performed in this paper can be used in radar systems Metzger et al. [26], parallel implementations of certain branch and bound algorithms Gedron and Crainic [15], and for solution of irregular scientific problems in large scale parallel machines Das et al. [10].

5 Conclusions and Future Work

This paper describes an algorithm and implementation for parallel pairing generation. Since both use dynamic domain decomposition methods to parallelize this problem, a novel load balancing scheme is created and evaluated experimentally. The communication abstraction used in implementing this scheme is termed an *active interchange*, because it permits load balancing to proceed concurrently with each parallel process' problem solving activities. The resulting speedup obtained is near linear even on a large number of processors (up to 160 processors). This speedup is obtained in part by reducing the communication overheads experienced in our cluster computing environment, by judicious use of both message passing with threads-based parallelism, thereby also exploiting the mixed distributed and shared memory computing environment prevalent in today's cluster computers.

We also address algorithm and implementation scalability to hundreds of processors, by development of and experimentation with the (MPA) pairing generation algorithm. We posit that additional performance improvements may be obtained with MPA for larger machines than those available to us. Parallel pairing generation has already been used successfully in the development of new approaches to airline crew scheduling Klabjan [21].

We believe that the parallel pairing generation methods demonstrated in this paper can also be used in branch-and-price algorithms and can open up a new research area in this context.

6 Acknowledgments

The authors would like to thank Intel Corporation, who funded the parallel computing environment via a large scale equipment grant; ILOG for providing the CPLEX solver; and United Airlines for providing the input data.

We would also like to thank Profs. Ellis Johnson and George Nemhauser for their guidance concerning pairing generation algorithms.

References

[1] R. Anbil, C. Barnhart, E.L. Johnson, and L. Hatay, 1994. A Column Generation Technique for the Long-Haul Crew Assignment Problem, in *Optimization in Industry II*, T. Ciriani and R. Leachman, eds., Jonh Wiley & Sons Ltd, New York, NY, 7–24.

[2] R. Anbil, E.L. Johnson, and R. Tanga, 1992. A Global Approach to Crew Pairing Optimization, *IBM Systems Journal* 31, 71–78.

[3] B.A. Blake, W. Bo, and K. Schwan, 1989. Global Data and Control in Multicomputers: Operating System Primitives and Experimentation with a Parallel Branch-and-Bound Algorithm, in *Concurrency and Experience*, John Wiley & Sons, New York, NY.

[4] B.A. Blake and K. Schwan, 1991. Experimental Evaluation of a Real-Time Scheduler for a Multiprocessor System, *IEEE Transactions on Software Engineering* 17, 34–44.

[5] W. Bo and K. Schwan, 1990. Topologies-Distributed Objects on Multicomputers, *ACM Transactions on Computer Systems* 8, 111–157.

[6] H.D. Chu, E. Gelman, and E.L. Johnson, 1997. Solving Large Scale Crew Scheduling Problems, *European Journal of Operational Research* 97, 260–268.

[7] D. Cronk, M. Haines, and P. Mehrotra, 1994. On the Design of Chant: A Talking Threads Package, in *Proceedings of Supercomputing*, Washington, D.C.

[8] D.E. Culler, T. Von Eicken, S.C. Goldstein, and K.E. Schauser, 1992. Active Messages: a Mechanism for Integrated Communication and Computation, in *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia.

[9] D.E. Culler and S.S. Lumetta, 1988. Managing Concurrent Access for Shared Memory Active Messages, in *Proceeding of the 12th International Parallel Processing Symposium*, Los Alamitos, CA.

[10] R. Das, M. Uysal, J. Saltz, and Y. Hwang, 1994. Communication Optimization for Irregular Scientific Computations on Distributed Memory Architectures, *Journal of Parallel and Distributed Computing* 22, 462–479.

[11] J. Eckstein, 1994. Parallel branch-and-bound methods for mixed integer programming, *SIAM News* 27, 12–15.

[12] J. Eckstein, 1997. How much communication does Parallel branch and bound need, *INFORMS Journal of Computing* 9, 15–29.

[13] J. Eykholt, S. Kleiman, J. Voll, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner, 1992. Symmetric Multiprocessing in Solaris 2.0, in *37th Annual IEEE International Computer Conference COMPCOM SPRING 1992*, San Francisco, CA.

[14] I. Foster, C. Kesselman, and S. Tuecke, 1996. The Nexus Approach to Integrating Multithreading and Communication, *Journal of Parallel and Distributed Computing* 40, 35–48.

[15] B. Gedron and T.G. Crainic, 1994. Parallel branch-and-bound algorithms: Survey and Synthesis, *Operations Research* 42, 1042–1066.

[16] G.W. Graves, R.D. McBride, I. Gershkoff, D. Anderson, and D. Mahidhara, 1993. Flight Crew Scheduling, *Management Science* 39, 736–745.

[17] A. Hac and X. Jin, 1990. Dynamic Load Balancing in a Distributed System Using a Sender-Initiated Algorithm, *Journal of Systems Software* 11, 79–94.

[18] R.N. Heederik, L.O. Hertzberger, B.J. Overeinder, and P.M.A. Sloot, 1996. A Dynamic Load Balancing System for Parallel Cluster Computing, *Future Generation Computer Systems* 12, 101–115.

[19] K.L. Hoffman and M. Padberg, 1993. Solving Airline Crew Scheduling Problems by Branch-and-Cut, *Management Science* 39, 657–682.

[20] J. Hu, 1996. *Solving Linear Programs Using Primal-dual Subproblem Simplex Method and Quasi-explicit Matrices*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA.

[21] D. Klabjan, 1999. *Topics in Airline Crew Scheduling and Large Scale Optimization*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA.

[22] S. Kleiman, B. Smallders, D. Stein, and D. Shah, 1992. Writing Multithreaded Code in Solaris, in *37th Annual IEEE International Computer Conference COMPCOM SPRING*, San Francisco, CA.

[23] L. Letovsky, 1997. *Airline Operations Recovery: An Optimization Approach*, Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA.

[24] A. Mainwaring and L.W. Tucker, 1994. CMMD: Active Messages on the CM-5, *Parallel Computing* 20, 481–496.

[25] Message Passing Interface Forum, 1995. *The MPI Message Passing Standard*, Knoxville, TN.

[26] R.C. Metzger, B. VanVoorst, L.S. Pires, R. Jha, W. Au, M. Amin, D.A. Castanon, and V. Kumar, 1996. C31 Parallel Benchmark Suite-Introduction and Preliminary Results, in *Supercomputing 96*, Pittsburgh, PA.

[27] B. Nickols, D. Buttlar, and J.P. Farrel, 1996. *Pthreads programming*, O'Reilly & Associates, Sebastopol, CA.

[28] M.J. Quinn, 1994. *Parallel Computing: Theory and Practice*, McGraw-Hill, New York, NY.

[29] A.P. Reeves and M.H. Willebeek-Lemair, 1993. Strategies for Dynamic Load Balancing on Highly Parallel Computers, *IEEE Transactions on Parallel and Distributed Systems* 4, 979–993.