# Airline Crew Pairing Generation in Parallel*

***Diego Klabjan*†, *Karsten Schwan*‡**

## 1 Introduction

Airline crew scheduling is the problem of assigning crew itineraries or pairings to flights. 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 and subsets correspond to pairings. 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 pairings ranges from 2 million for small problems, to 20 million, to 50 million and above for large problems. 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. 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. Therefore, speeding up the pairing generation process is a necessary step toward reducing crew scheduling time.

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

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

†Department of Mechanical and Industrial Engineering, University of Illinois, Urbana, IL, klabjan@uiuc.edu

‡College of Computing, Georgia Institute of Technology, Atlanta, GA, schwan@cc.gatech.edu

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. The detailed discussion of active interchange is given in [2]. 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. 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. 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.

# 2 Parallel Algorithm for Pairing Generation

## 2.1 Algorithm Description

Crew bases are stations where crews are stationed. Every pairing must start and end at a crew base. 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. 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, denoted by SA, scans all the starting duties and for each starting duty $d$ it performs DFS($d$).

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 2.1 (assuming there are $p$ processes). A major iteration of the algorithm is comprised of Steps 8-14.

**Algorithm 2.1** 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:  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 $D_i$.
14:  **end while**
15: **end for**

## 2.2 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, [1]. 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.

## 2.3 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 line 13 in (PA), 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 addressed 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 7-15 with Algorithm 2.2.

**Algorithm 2.2** Modified Parallel Algorithm (MPA)

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

# 3 Computational Experiments

## 3.1 Implementation Issues

The parallel implementations use the MPI message passing interface, [3], MPICH implementation Version 1.0, developed at Argonne National Labs, and a portable threads standard is used, Pthreads (POSIX threads) 4, [4].

Our first implementation was exclusively a message passing one. Results obtained using only MPI were poor and are not reported in this paper. All the

implementations use a minimum of two threads per process. One thread, the working thread, does the actual 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 this mechanism 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.

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 $\mathrm{P}[X = i] = cpu_i \cdot p_i / \sum_{j=1}^{p} (cpu_j \cdot p_j)$. 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 step 13 of PA to reflect the above observation. The modified step reads

13: 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.

Further details on the implementations are given in [2].

## 3.2 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.

### 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 (mixed MPI/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 pure MPI code. These

results are due to the fact that the use of threads reduces communication overhead.

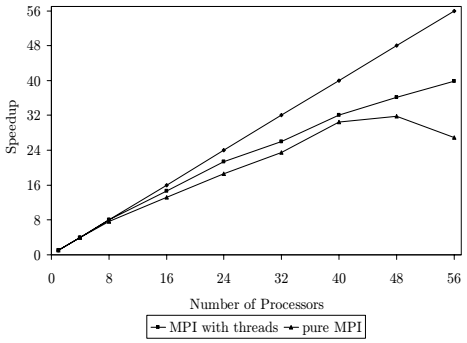


**Figure 1.** *The comparison of pure MPI implementation and mixed MPI/threads implementation*

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 drop offs in speedup toward the end of MPI program runs, where too many idle processors are requesting new work.

### Speedup

The results depicted in Figure 2 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 on the left 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 computation 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 the left figure vs. the right figure. 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,

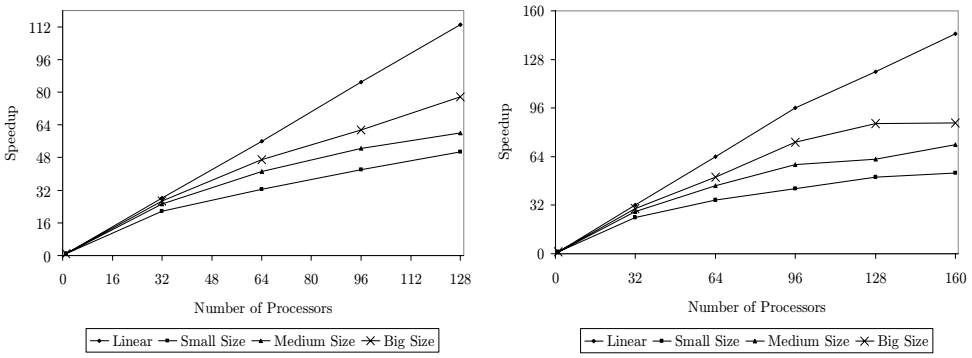


**Figure 2.** *Left figure: Half 300 MHz nodes, half 200 MHz nodes. Right figure: As many 300 MHz nodes as possible.*

actual execution times do not differ much when using 200 vs. 300 MHz nodes, with actual execution times being identical 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 3.2 below.

### 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 16 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 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 3, 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.

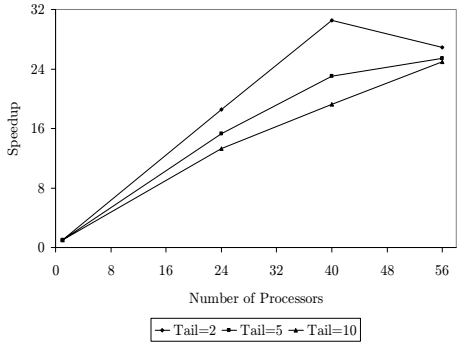


**Figure 3.** *Tail effect*

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 heuristic. 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 based on experimental computational runs and then using regression analysis to determine the 'best fit' values.

We were using the following formulas: $b_i = i + \frac{F \cdot n}{t^2}, tail = \frac{F_1}{td}, tp = \frac{F_2 \cdot p}{F_3 + td}$. The parameter values for our hardware are $F = 3.5, F1 = 0.96, F2 = 0.02$, and $F3 = 1.7$.

| | PA | MPA |
|---|---|---|
| p=64 | 570 | 497 |
| | 482 | 480 |
| | 535 | 500 |
| | 540 | 490 |
| | 1245 | 1145 |
| p=50 | 570 | 500 |

**Table 1.** *Computational comparison of PA and MPA*

The results are presented in Table 1. 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 would be more significant for problems with worse computation/communication ratios than the ones used in our work.

## 4 Acknowledgments

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

# Bibliography

[1] B. Blake and K. Schwan, *Experimental evaluation of a real-time scheduler for a multiprocessor system*, IEEE Transactions on Software Engineering, 17 (1991), pp. 34–44.

[2] D. Klabjan and K. Schwan, *Airline crew pairing generation in parallel*, Tech. Rep. TLI/LEC-99-02, Georgia Institute of Technology, 1999.

[3] Message Passing Interface Forum, *The MPI Message Passing Standard*, 1995. Available from http://www.mpi-forum.org.

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