Static Multi-processor Scheduling with Ant Colony Optimisation & Local Search

Graham Ritchie

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2003

Abstract

Efficient multi-processor scheduling is essentially the problem of allocating a set of computational jobs to a set of processors to minimise the overall execution time. There are many variations of this problem, most of which are NP-hard, so we must rely on heuristics to solve real world problem instances. This dissertation describes several novel approaches using the ant colony optimisation (ACO) meta-heuristic and local search techniques, including tabu search, to two important versions of the problem: the static scheduling of independent jobs onto homogeneous and heterogeneous processors.

Finding good schedules for jobs allocated on homogeneous processors is an important consideration if efficient use is to be made of a multiple-CPU machine, for example. An ACO algorithm to solve this problem is presented which, when combined with a fast local search procedure, can outperform traditional approaches on benchmark problems instances for the closely related bin packing problem. The algorithm cannot compete, however, with more modern specialised techniques.

Scheduling jobs onto hetereogeneous processors is a more general problem which has potential applications in domains such as grid computing. A fast local search procedure for this problem is described which can quickly and effectively improve solutions built by other techniques. When used in conjunction with a well-known heuristic, Min-min, it can find shorter schedules on benchmark problems than other solution techniques found in the literature, and in significantly less time. A tabu search algorithm is also presented which can improve on solutions found by the local search procedure but takes longer. Finally a hybrid ACO algorithm which incorporates the local and tabu searches is described which outperforms both, but takes significantly longer to run.

Acknowledgements

I would like to thank my supervisor, John Levine, for his help and advice throughout this project. I would also like to thank Tracy Braun and Howard Siegel for sharing their results with me, and Jon Schoenfield for providing lower bound proofs for some test problems used in this dissertation.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Graham Ritchie*)

Table of Contents

Chapter 1

Introduction

The problem of efficiently scheduling computational jobs on multiple processors is an important consideration when attempting to make effective use of a multi-processor system such as a computational grid or a multiple CPU machine. The multi processor scheduling problem (MPSP) can be formulated in several ways, taking into account different constraints. Two important variations include scheduling independent jobs onto homogenous and heterogeneous processors.

The homogeneous MPSP is defined here as the problem of scheduling a set of n independent jobs each of which has an associated running time, onto a set of m identical processors such that the overall execution time is minimised. The heterogeneous MPSP is identical except that the rule that the processors must be identical is relaxed, and it is assumed that each job can take a different time to execute on each of the available processors.

There is a distinction to be made between *dynamic* and *static* multi-processor scheduling. Dynamic scheduling deals with jobs as and when they arrive at the scheduler and can deal with changing numbers of processors. This dissertation is, however, only concerned with static scheduling, and so all necessary information about the jobs and processors, e.g. running times, and the number and nature of the processors, are assumed to be known *a priori*.

In most of its practical formulations, including the two just described, the MPSP is known to be NP-hard [Garey and Johnson, 1979], so exact solution methods are unfeasible for most problem instances and heuristic approaches must therefore be em-

ployed to find solutions. Heuristics are by their nature non-exhaustive, and so there is often scope for different approaches to better previous solution methods according to some metric or other (e.g. execution speed, quality of solutions etc.) Traditional approaches to the MPSP are as varied as the different formulations of the problem, but include fast, simple heuristics (see e.g. [Braun et al., 2001]), tabu search (e.g. [Thiesen, 1998]), evolutionary approaches (e.g. [Corcoran and Wainwright, 1994]), and modern hybrid algorithms that consolidate the advantages of various different approaches (e.g. [Alvim et al., 2002]).

In this dissertation I describe two closely related approaches to these two versions of the MPSP combining the Ant Colony Optimisation (ACO) meta-heuristic with local search methods, including tabu search. The ACO meta-heuristic was first described by Dorigo in his PhD thesis [Dorigo, 1992], and was inspired by the ability of real ant colonies to efficiently organise the foraging behaviour of the colony using external chemical *pheromone* trails acting as a means of communication. ACO algorithms have since been widely employed on many NP-hard combinatorial optimisation problems, including many problems related to the MPSP such as bin packing [Ducatelle, 2001, Levine and Ducatelle, 2003] and job shop scheduling [van der Zwaan and Marques, 1999]. However, they have not previously been applied to the two MPSP variations described above. Local search methods encompass a wide array of approaches to many optimisation problems, and have been shown to be very effective when used in combination with an ACO approach [Dorigo and Stützle, 2002].

The approach described in this dissertation for the homogeneous problem performs well in terms of the quality of solutions found when compared, using benchmark problems, to traditional solution techniques found in the literature, but can often take longer to run. The approach is successful, but cannot outperform state of the art algorithms.

Experimental results for the heterogeneous problem indicate that the approach described here is particularly well suited to this problem. A range of techniques are proposed which can outperform other solution techniques found in the literature both in terms of the quality of solutions found and the time taken to find them.

There are of course other possible variations of the version of the MPSP than those studied here, for example the problem of scheduling jobs with precedence constraints,

and a discussion of possible further applications is provided in chapter 7. It is hoped however that the fundamental ideas presented here could be applied to such related problems.

It should be noted that although the terminology of computational jobs and processors is used throughout this dissertation, the problem can be considered a general scheduling problem. The approaches discussed here could therefore be applied to other scheduling tasks, such as scheduling of industrial jobs in a factory. As long as the salient features of the problem are clearly identifiable, such as a job has a definite running time, or a machine can process jobs at a certain speed, the techniques described here could be applied.

This dissertation is organised as follows. Chapter 2 introduces the ACO metaheuristic in detail, discussing its biological inspiration and subsequent technical evolution, and provides an introduction to the local search concepts used in this study. Chapter 3 provides a more detailed explanation of the homogeneous MPSP studied in this dissertation and discusses some important existing solution techniques. Chapter 4 then describes how ACO and local search techniques were applied to the homogeneous problem, and provides experimental results. The heterogeneous problem is introduced in detail in chapter 5 along with some current solution techniques. Chapter 6 describes how ACO and local search were applied to this second problem and provides experimental results. Chapter 7 concludes with a summary of the project, its successes and shortcomings and discusses possible future work on approaching other variations of the MPSP with ACO techniques.

Chapter 2

An introduction to ant colony optimisation and local search techniques

The ant colony optimisation meta-heuristic (ACO) was first described by Marco Dorigo in his PhD thesis, and was initially applied to the travelling salesman problem (TSP). It has since been applied to many other combinatorial optimisation problems, as well as other tough problems (see [Dorigo and Stützle, 2002] for a review). It is inspired by the abilities of real ants to find the shortest path between their nest and a food source. This chapter provides an introduction to ACO and its biological inspiration, and describes the mechanisms the approach uses to find good solutions to tough problems.

This chapter also provides a brief introduction to local search methods, including tabu search (TS), and discusses the benefits of combining ACO and local search.

2.1 Biological inspiration

The ACO approach was inspired by the foraging behaviour of real ants. Many species of ant deposit a chemical *pheromone* trail as they move about their environment, and they are also able to detect and follow pheromone trails that they may encounter. This

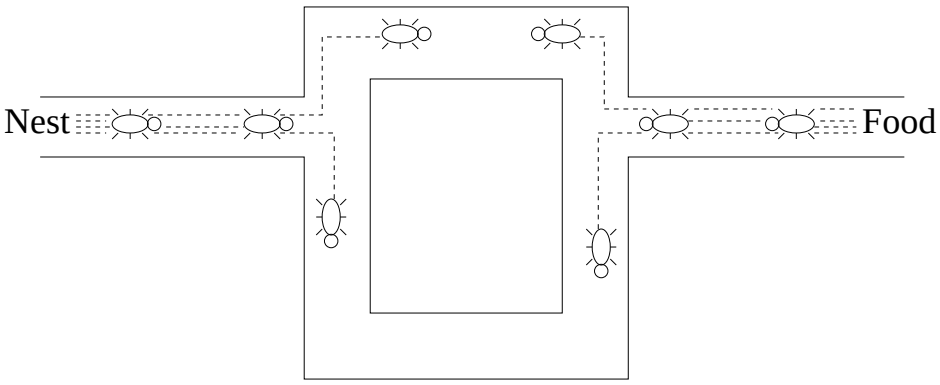

Figure 2.1: An illustration of the original (biological) ant experiment. Figure from [Levine and Ducatelle, 2003].

process is known as *recruitment*.

Deneubourg et al. (described in detail in [Bonabeau et al., 1999]) have shown that the members of the ant species *Linepithema humile* use recruitment to efficiently organise the foraging behaviour of the ant colony. They showed this using an simple experiment in which two bridges of different lengths were placed between the ants' nest and a source of food, a schematic view of the experiment is shown in figure 2.1. Initially the ants picked their route apparently at random, and about 50% of the ants used each bridge. However as time passed, and the pheromone trails increased, the ants tended to favour the shortest route to the food. The pheromone trail increased on the shorter bridge faster than on the longer one because the ants using it to gather food would take a shorter time, and so more ants would cross it, and so more pheromone would be deposited. No single ant could derive this information, but the colony can and hence it is an example of 'swarm intelligence' (SI). It is also important to note that while the majority of the ants used the shorter bridge, throughout the experiment some ants continued to use the longer path - it seems that they pick a route probabilistically with respect to the amount of pheromone. This is useful, however, because it means that they ants do not converge on a single path. This allows them to dynamically adjust their behaviour in response to possible obstructions in their route - if the shorter bridge is removed, the ants quickly start to use the alternative route.

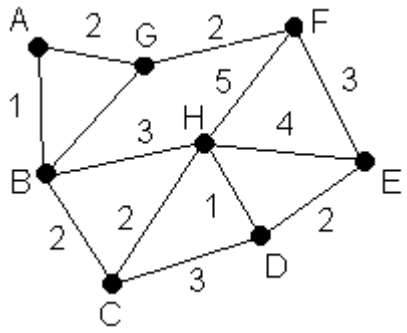

Figure 2.2: A simple TSP problem. The lettered nodes represent cities and the numbered arcs represent the distance between any two cities.

2.2 From inspiration to algorithm

The first ACO algorithm was applied to the travelling salesman problem (TSP). This is the problem of finding the shortest tour visiting all the nodes of a fully-connected graph, the nodes of which represent locations, and the arcs represent a path with an associated cost (normally assumed to be distance). An example problem for the TSP is shown in figure 2.2. This problem has a clear analogy with the shortest path finding ability of real ants, and is also a widely studied NP-hard combinatorial optimisation problem.

Dorigo applied ACO to the TSP with his 'Ant System' (AS) approach ([Dorigo, 1992]). In AS each (artificial) ant is placed on a randomly chosen city, and has a memory which stores information about its route so far (a partial solution), initially this is only the starting point. Setting off from its starting city an ant builds a complete tour by probabilistically selecting cities to move to next until all cities have been visited. While at city i, an ant k picks an unvisited city j with a probability given by equation 2.1.

$$p_k(i,j) = \frac{[\tau(i,j)]^{\alpha}.[\eta(i,j)]^{\beta}}{\sum_{l \in N_i^k}[\tau(i,l)]^{\alpha}.[\eta(i,l)]^{\beta}} \tag{2.1}$$

In this equation $\eta(i,j) = \frac{1}{d(i,j)}$, where $d(i,j)$ is the distance between cities i and j, and represents heuristic information available to the ants. N_i^k is the 'feasible' neigh-

bourhood of ant k, that is all cities as yet unvisited by ant k. $\tau(i,j)$ is the pheromone trail value between cities i and j. α and β are parameters which determine the relative influence of the heuristic and pheromone information, if α is 0 the ants will effectively be performing a stochastic greedy search (see section 2.3) using the 'nearest-neighbour' heuristic, if β is 0 then the ants use only pheromone information to build their tours.

After all the ants have built a complete tour the pheromone trail is updated according to the *global update rule* defined in equation 2.2. In this equation ρ is a pheromone evaporation parameter which decays the pheromone trail (and thus implements a means of 'forgetting' solutions which are not reinforced often), and m is the number of ants.

$$\tau(i,j) = \rho.\tau(i,j) + \sum_{k=1}^{m} \Delta\tau_k(i,j) \tag{2.2}$$

The specific amount of pheromone, $\Delta\tau_k(i,j)$, that each ant k deposits on the trail is given by equation 2.3. In this equation L_k is the length of ant k's tour. This means that the shorter the ant's tour, the more pheromone will be deposited on the arcs used in the tour, and these arcs will thus be more likely to be selected in the next iteration.

$$\Delta\tau_k(i,j) = \begin{cases} \frac{1}{L_k} & \text{if arc}(i,j)\text{is used by ant } k \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

The algorithm iterates through each of these stages until the termination criteria (e.g. a maximum number of iterations) are met.

The results from AS were encouraging, but it did not improve on state of the art approaches. This original system has since been adapted in various ways, including using an *elitist strategy* which only allows the iteration or global best ant to leave pheromone (known as Max-Min AS, and explained in detail in [Stützle and Hoos, 2000]), and allowing the ants to leave pheromone as they build the solution (see [Dorigo and Stützle, 2002] for details).

Since AS and its adaptations the ACO approach has been modified and applied to many different problems, and the implementation details have moved some way from the original biological analogy. [Dorigo and Stützle, 2002] provides a useful distilla-

tion of the ideas of various approaches to implementing ACO algorithms, and describes the main features any ACO approach must define for a new problem, the five main requirements are identified below:

- A heuristic function $\eta()$, which will guide the ants' search with problem specific information.
- A pheromone trail definition, which states what information is to be stored in the pheromone trail. This allows the ants to share information about good solutions.
- The pheromone update rule, this defines the way in which good solutions are reinforced in the pheromone trail.
- A fitness function which determines the quality of a particular ant's solution.
- A construction procedure that the ants follow as they build their solutions (this also tends to be problem specific).

Although the ACO method is a fairly novel approach, it can be viewed as something of an amalgam of other techniques. The ants essentially perform an adaptive greedy search of the solution space. Greedy search methods have been used for many problems, and are often good at finding reasonable results fast (greedy search is discussed in more detail in section 2.3). The greedy search takes into account information provided in the pheromone trail, and this information is reinforced after each iteration. This is a form of 'reinforcement learning' (RL) where solutions are reinforced proportionally to their value. RL (in various guises) has been used to good effect in many other hard problems (for more information on RL see [Sutton and Barto, 1998]). Because a population of ants is used, ACO also has similarities with evolutionary computation approaches, such as genetic algorithms (GAs). GAs similarly use a population of solutions which share useful information about good solutions, and probabilistically use this information to build new solutions. They do this in a rather different way however - a GAs 'stores' its information in the population at each iteration whereas the ants store it in the pheromone trail (for more details of many evolutionary strategies see [Mitchell, 1998]). So, while the specific details of an ACO algorithm are new, and

the techniques used are combined in a novel way, the ACO approach implicitly uses several proven problem solving strategies.

2.3 Local search mechanisms

Local search techniques have long been used to attack many optimisation problems (see e.g. [Russell and Norvig, 1995] for an overview). The precise method used is almost entirely problem dependent, but the important concept is that a solution to the problem in hand has an identifiable solution *neighbourhood*. The neighbourhood of a solution is generally defined as all those solutions which differ from the original solution by a single 'step'. The definition of such a step is, again, problem dependent (see below for an example for the TSP). The simplest implementation of a local search mechanism, known as *iterative improvement*, firstly creates an initial solution by some means (possibly just generating one at random). The algorithm then checks through some or all of the neighbours of the initial solution looking for an improved solution. If an improved solution is found then the current solution is replaced with it and the process repeats until no further improvement can be found. If the solution space is viewed as a landscape then this iterative process can be seen as 'hill-climbing' to local peaks, or optima. There are two common means of implementing iterative improvement algorithms known as *first ascent* and *best ascent*. In the former the first neighbour which is found to be an improvement is selected, whereas the latter searches through all possible neighbouring solutions and then selects the one which offers the most improvement.

As an example, in the TSP a simple neighbourhood definition used is that of the 2-opt neighbourhood in which a solution is defined as a neighbour if it differs by at most 2 arcs (this can of course be generalised to deal with k arcs, and is thus generally known as the *k-opt* approach). An example of a solution and an improved neighbour is given in figure 2.3.

A common variation on the simple iterative improvement algorithm is, rather than rigidly selecting either the first improved solution or the best neighbouring solution, is to pick a next solution probabilistically in proportion to the amount of improvement a

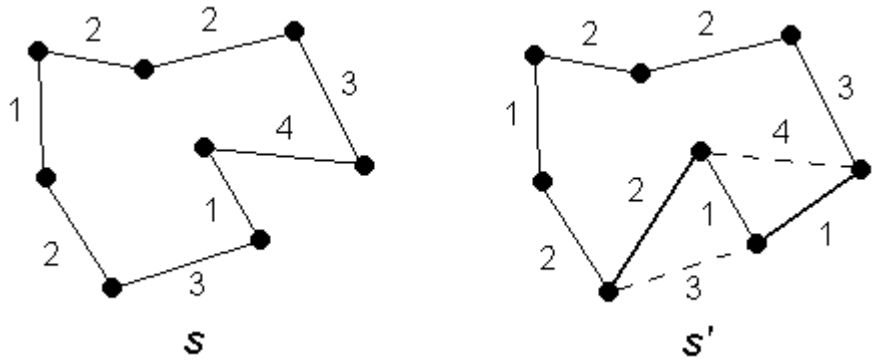

Figure 2.3: Illustration of a 2-opt neighbour in the TSP. s' is a improved neighbour of s because it varies by 2 arcs, and its tour length is 4 less than s.

neighbouring solution provides. This is known as a *stochastic* local search, and allows the search a little more flexibility which, if repeated, can help to find better solutions by exploring more of the search space than the simpler techniques.

Local search techniques are simple and fast, and can be very effective for some problems, but they do not fare very well in 'bumpy' solution landscapes - those which have multiple 'peaks'. The local search can quickly take a solution to a local optimum, but it may be that this is not the global optimum. If the search is repeated the stochastic approach can help to resolve this to an extent, but is still liable to the problem in solution spaces encountered in many real problems.

2.3.1 ACO and local search

Many researchers (e.g. [Dorigo and Stützle, 2002] and [Levine and Ducatelle, 2003]) have found that coupling local search mechanisms with ACO algorithms can produce very good results. The usual approach is to use the ants to generate a number of solutions, and then each of these solutions is then taken through a local search phase, the (hopefully) improved solutions are then used in the pheromone update stage. This coupling can therefore be seen as a form of Lamarckian search.

On reflection this coupling seems very complimentary, the ants perform a rather course grained search of the whole solution space [Dorigo and Stützle, 2002], and so their solutions are very amenable to optimisation. Local search techniques, on the other

hand, are very good at finding local optima, but it is difficult to define useful starting points for the local search. Approaches such as random restart hill climbing which simply restart a local search at random points in the solution space have been used to some effect, for example the WalkSAT algorithm described in [Selman et al., 1993] which is very effective at finding good solutions for satisfiability problems. However I feel that such approaches are inherently rather inefficient because information about good regions of the search space is effectively thrown away when the algorithm restarts again from a random region in the search space. It would be much preferable to use decent solutions to begin with. This is exactly what the ants can provide: solutions which have been informed and led by both heuristic information and shared information about good solutions found previously (in the pheromone trail).

2.3.2 Tabu search

Tabu search (TS) is a meta-strategy for guiding known heuristics to help then avoid local optima. It was first developed by Fred Glover [Glover and Laguna, 1997], and has since been applied to many difficult combinatorial optimisation problems in a number of different areas. A TS approach has four main elements ([Thiesen, 1998]):

- A local search strategy.
- A mechanism to discourage a return to a recently visited solution - the ‘tabu list’.
- A ‘tabu tenure’ policy used to determine how long a solution will remain in the tabu list.
- A mechanism to alter the search path when no improvement has been made for some time.

The local search is usually a simple greedy strategy which finds an improved solution in the immediate neighbourhood of a current solution. In order to stop the search repeating itself, recently visited solutions are added to the tabu list, and if a solution found by the local search already exists in the tabu list it is forbidden, or ‘taboo’, and an alternative, non-tabu, solution will be used. To ensure the search doesn’t quickly

exclude all neighbours, solutions are only held in the tabu list for some period of time - the tabu tenure. The mechanism used to alter the search path varies from problem to problem, but a common approach is to simply pick a random solution in the neighbourhood of the current solution. TS thus sometimes allows non-improving solutions to be used for the sake of avoiding local optima.

TS clearly relies heavily on a good local search, but the intuition behind it is that it can guide the path of the local search using a simple form of short-term memory (the tabu list) helping it to avoid uninteresting, previously explored areas in the search space.

2.3.3 ACO and tabu search

TS is essentially a smarter local search, and so the previous arguments about why local search is a useful addition to an ACO algorithm also apply here. Although 'pure' TS approaches that are used for real problems generally involve a large number of iterations and use fairly complex tabu list structures and tenure policies, along with other additions such as diversification etc. (see [Glover and Hübscher, 1994] for one example), a fairly simple TS will often still beat a 'pure' iterative improvement algorithm, and when combined with an ACO approach, the ants can be viewed as a diversification component for the TS. The ants are used to generate promising initial solutions and then TS is used to try to improve on these solutions.

2.4 Summary

This chapter has provided a brief overview of the problem solving concepts used to tackle the MPSP in this dissertation. The ACO meta-heuristic has been described and contrasted with other meta-heuristic approaches, and I have argued that combining ACO with local search techniques appears to be a very complimentary partnership.

Chapter 3

Homogeneous multi-processor scheduling

As described earlier the MPSP is in general the problem of allocating a set of n jobs to a set of m processors, processing all the jobs in the quickest possible time. The first variation of this problem studied in this dissertation, scheduling jobs onto identical, or homogeneous, processors is introduced in this chapter, along with an overview of solution techniques found in the literature.

3.1 A formal definition

The homogeneous MPSP is defined as the problem of scheduling a set J of n independent jobs $J = \{j_1 \dots j_n\}$, each of which have an associated running time $t(j_i)$ onto a set P of m identical processors, $P = \{p_1 \dots p_m\}$, such that all the jobs are completed as quickly as possible, i.e. minimising the makespan of the solution. A lower bound on the makespan of an optimal solution for this problem is simply calculated as shown in equation 3.1.

$$\left\lceil \frac{\sum_{i=1}^{n} t(j_i)}{m} \right\rceil \tag{3.1}$$

Although the lower bound is simple to calculate, this problem has been shown to

be NP-hard (it is a reformulation of problem SS8 in Garey and Johnson's review of NP-hard problems [Garey and Johnson, 1979]) and so only limited problem instances can be optimally solved with exact algorithms in reasonable time. Heuristic algorithms therefore appear to offer the most promising approach.

Finding good, and preferably optimal, solutions to this problem is important if we are to make efficient use of a homogeneous multi-processor computing system, such as a multi-CPU machine. Only static scheduling is considered here, but this is useful if, for example, access is available to a multi-processor super-computer for a limited time, but the jobs are known in advance. Identifying and solving such problems is also important in related scheduling problems that occur in such domains as planning. For example [Fox and Long, 2001] use the FFD heuristic (which is sub-optimal, see below for a description) to solve a MPSP when their planning system *STAN4* identifies a MPS sub-problem of this type in planning problems. As mentioned briefly before, the MPSP is also part of a larger class of scheduling problems which have applications in industrial scheduling, resource allocation etc.

3.2 Comparison with the BPP

The MPSP as formulated here is closely related to another NP-hard combinatorial optimisation problem, the bin packing problem (BPP). The BPP is defined as the problem of packing a given set I of n items $I = \{i_1 \dots i_n\}$ each of which is a certain size $s(i_j)$ into as few bins of a fixed capacity C as possible. A lower bound on the number of bins for a given problem is shown in equation 3.2.

$$\left\lceil \frac{\sum_{j=1}^{n} s(i_j)}{C} \right\rceil \tag{3.2}$$

If the items are viewed as jobs and the bins as processors, then the MPSP can be seen as equivalent to a BPP where the bin capacity C is equal to the lower bound on the makespan as given by 3.1, and the goal is to find a solution such that the number of bins equals the number of available processors m. However if no such solution exists (or is very hard to find) then the two problems diverge, as the BPP would require that

another bin is used to pack the left over items, while the MPSP would not allow this and would extend the makespan instead.

Solving BPP problems is of direct interest in many industrial fields where resources must be used as efficiently as possible, for example as few lorries as possible should be used to transport stock. The BPP is also part of a class of related problems, which also include the cutting stock problem (CSP) which have applications in many manufacturing domains, VLSI design, budgeting etc.

3.3 Current solution techniques

Because the MPSP and the BPP approaches are so similar, an algorithm that works on one can often be successfully adapted to the other with only minor modification. This section therefore describes some current solution techniques for both the MPSP variation studied here, and the BPP.

3.3.1 MPSP solution techniques

3.3.1.1 Greedy heuristics

Several simple and fast greedy heuristics exist for the MPSP. A simple one allocates the jobs in arbitrary order to the least loaded processor (LLP). A better variation of LLP, known as LPT, is to sort the jobs into descending order of running time first and then run LLP. The best of the greedy heuristics is ‘first-fit decreasing’ (FFD), which combines some of the ideas of both of these. This algorithm firstly establishes the optimal makespan of this problem, using equation 3.1, and then sorts the set of jobs to be scheduled into decreasing order of running times. Each job is then taken in turn from this list and allocated to the first processor onto which it will fit without exceeding the optimal makespan. If the job will not fit onto any processor without exceeding this value it is allocated to the least loaded processor. Another similar heuristic that is used is ‘best-fit decreasing’ (BFD). This algorithm similarly allocates each job in order of decreasing running time, but instead of allocating it to the first processor on which it will fit, it is allocated to the processor on which it will fit ‘best’, i.e. the processor with

a gap in its schedule that most closely matches the running time of the job. BFD is clearly more complex than FFD, but, surprisingly, in tests on the benchmark problem instances used in this dissertation it performs slightly worse.

FFD seems to work well because it schedules larger jobs early on a processor, and then fills up the time left with the smaller jobs - this means that big jobs don't 'stick out' too much, and jobs of different sizes are usefully combined on a processor. Although it often exceeds the optimum makespan, it often finds well-balanced solutions within a few time units of the optimum.

3.3.1.2 Tabu search

[Thiesen, 1998] proposes a tabu search (TS) specifically for this version of the MPSP (but compares it with BPP techniques). Thiesen is mainly concerned with investigating different TS strategies and so uses several different techniques, but the main steps in his approach are given below.

- Create an initial *solution* and an empty tabu list of a given size, and set the *best solution* to be the initial solution.
- Repeat the following steps until the termination criteria are met:
  - Add *solution* to the tabu list, if the tabu list is full then delete the oldest entry in the list. Sufficient information about entire solutions are efficiently stored in the list using a hash coding scheme (described in detail in section 6.7).
  - Find *new solution* by performing a single local neighbourhood search on *solution*, if *new solution* is better than the current *best solution*, *best solution* becomes *new solution*.
  - If no new solution was found, or *best solution* has not improved for some time, then generate *new solution* at random.
  - If *new solution* is not on the tabu list then set *solution* to be *new solution*.

Thiesen used the LPT heuristic to build his initial solution, and uses several different local searches, including probabilistic and greedy search, to generate his *new*

*solution*s. The simplest used was just to check whether any pair of jobs could be swapped between the busiest and least busy processor, or whether any job could be transferred from the busiest to the least busy processor, while reducing the total time of each. All such moves were compared, and the best one was chosen. Example termination criteria used were the number of iterations, the time taken, and the number of iterations since there was an improvement.

Thiesen tested his approach using his own randomly generated problems, and so no direct comparison can be made with other approaches (although he did compare results with an alternative tabu search approach applied to 'similar' problems). However his TS did find the optimal solution for all the problems he tested it on, which suggests that it significantly outperformed any simple greedy heuristic.

3.3.1.3 Evolutionary approaches

[Corcoran and Wainwright, 1994] describes a genetic algorithm (GA) for a slightly modified version of the MPSP studied here where the memory available on each processor and the memory requirement of each job is also taken into account. They use a several GA approaches, including a parallel island model and a simple serial GA. [Nath et al., 1998] present another GA which attempts to solve another slight variation on this MPSP, which tries to minimise another performance metric simultaneously: the 'total flowtime'. I have been unable to find any evolutionary approaches for the precise version of the MPSP dealt with here in the literature, but some approaches to the BPP are described in the next section.

3.3.2 BPP solution techniques

3.3.2.1 Greedy heuristics

Many of the greedy heuristics used for the MPSP can be applied to the BPP with only minor modifications. FFD for the BPP sorts the items into decreasing order of size, and then, very similar to previously, allocates each item in order to the first bin into which it will fit without exceeding the capacity constraint. If it cannot fit into any existing bin a new one is 'opened'. BFD simply allocates each item, in decreasing order of size

again, to the bin in which it fits best. Both FFD and BFD's worst case result for the BPP has been proved to be $\frac{11}{9}OPT + 4$, where OPT is the optimum number of bins for a given problem.

3.3.2.2 Martello & Toth's MTP algorithm

Such simple heuristics are fast and fairly effective, but they rarely find the optimal solutions for larger problems. The standard reference until recently has been Martello and Toth's exact solution method, known as MTP [Martello and Toth, 1990]. This algorithm is slower but does provide significantly better results. The algorithm starts by applying a set of three bounding tests (L_1, L_2 and L_3 - as an example L_1 is simply the lower bound provided by equation 3.2) to find a lower bound on the number of bins required. Next the FFD and BFD heuristics, along with an alternative heuristic referred to as 'worst fit decreasing' (WFD) are applied to the problem to see if they can solve it at the lower bound. If these heuristics fail to find such a solution an exhaustive branch-and-bound search is applied until either a solution at the lower bound is found, or a specified number of backtracks have been used. MTP also attempts to reduce a problem instance using a reduction procedure, MTRP. This procedure works by identifying *dominating* bins. Given two bins B_1 and B_2, if there exists a subset $\{i_1 \dots i_n\}$ of B_2 and a partition $\{P_1 \dots P_n\}$ of B_1 where for each item i_j there is no larger corresponding P_j, then B_2 is said to dominate B_1 (after [Falkenauer, 1996]). This is illustrated in figure 3.1. This is a useful distinction because a solution with B_2 in it requires no more bins than one with B_1 in it. MTP uses this criterion and attempts to find bins which dominate *all* other bins in the solution. Such bins can therefore be guaranteed to be in the optimal solution, and so the items they contain can be removed from the problem to leave a reduced problem to solve. This approach is effective, but a complete search would run into exponential time, therefore only bins with three items or less are considered.

3.3.2.3 Evolutionary approaches

There have been several attempts to attack the BPP with evolutionary approaches, e.g. [Vink, 1997] and [Reeves, 1996]. The benchmark problem instances used in this dis-

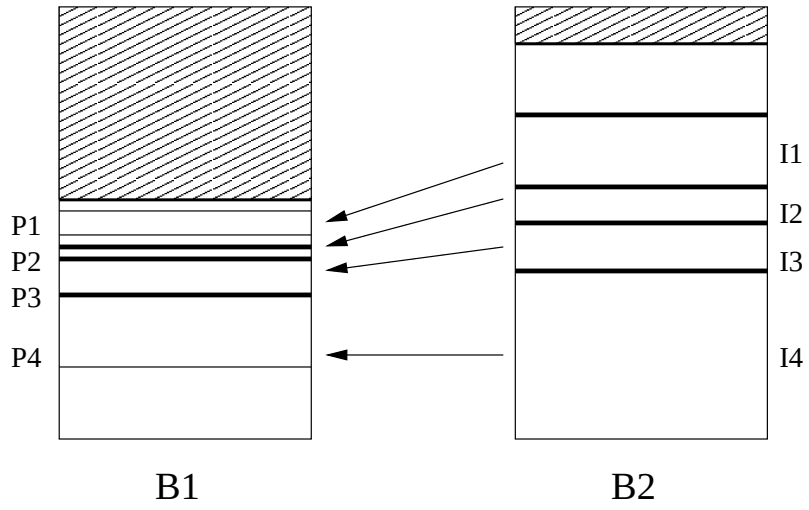

Figure 3.1: Bin $B2$ dominates $B1$. Figure after [Falkenauer, 1996].

sertation were first suggested in [Falkenauer, 1996], which describes a hybrid genetic algorithm (HGGA) for the BPP which can outperform Martello and Toth's approach on many problem instances.

HGGA uses standard genetic algorithm (GA) procedures to evolve good solutions to the BPP (se e.g. [Mitchell, 1998] for an introduction to GAs). The GA deals with whole bins, not just individual items. The crossover procedure essentially swaps bins between the two parents to produce offspring. This will mean, however, that some items will be repeated in some bins. The bins with repeats are removed from the solution, and the non-repeated items are considered 'free'. HGGA then tries to improve the solution by swapping free items with items currently in the bins which will better fill a bin. This procedure will (hopefully) leave smaller free items, which are then reinserted into the solution using FFD. Mutation works in a similar way - a few bins are randomly deleted and their items become free, and the same process is repeated.

[Ducatelle, 2001, Levine and Ducatelle, 2003] describe an ACO approach to the BPP, AntBin, which also outperforms MTP, and HGGA for some problem instances. The ants in AntBin follow the concept of the FFD heuristic, preferring to allocate big items early on, and also use information stored in a pheromone trail to probabilistically build solutions. The pheromone trail encodes the favourability of having two items of particular sizes together in a bin, and this trail is regularly reinforced with good

solutions found previously. AntBin also includes an effective local search which uses a similar method to HGGA to improve ant solutions: some number of bins are deleted, and their contents are then redistributed in the same way as HGGA. The combination of ACO and local search proved to be very effective, and when AntBin was combined with an iterated version of this local search [Levine and Ducatelle, 2003] it can solve many of the problems used to test HGGA to optimality.

3.3.3 A hybrid approach

The approach that performs the best on the benchmark BPP problem sets used to test both HGGA and AntBin (and used in this dissertation) is a hybrid approach described in [Alvim et al., 2002]. This approach combines several successful techniques, including a reduction procedure similar to Martello and Toth's, and a tabu search to produce a fast and robust algorithm. The main steps of the algorithms are outlined below. This approach mixes solution techniques to both the BPP and the MPSP. It is designed as a solution technique for the BPP, but essentially converts a BPP to an MPSP (step 2 below) and attempts to solve this optimally (steps 3-5), if an optimal solution is not found then the number of 'processors' is increased and the algorithm starts again (step 6).

1. *Reduction:* use Martello and Toth's reduction technique to eliminate some items and to establish the 'correct' bin for some others.

2. *Bounds:* calculate upper and lower bounds (by various means) to establish the optimal number of bins for the problem in hand.

3. *Construction:* use a greedy search to build a solution using the optimal number of bins (while relaxing the bin capacity constraint, and hence changing the problem to an MPSP). Three different greedy search techniques are tried and the best solution is used.

4. *Redistribution:* if the solution at this stage is not feasible (i.e. any bin exceeds the bin capacity), use load balancing strategies to improve the solution.

5. *Improvement:* if the current solution is still not feasible, perform a tabu search

on the current solution to attempt to find a solution which does not violate the capacity constraints (i.e. to reduce the makespan).

6. *Stopping criterion:* if the solution is feasible then stop, otherwise increase the lower bound by 1 and repeat the process beginning at the construction phase.

3.4 Summary

In this chapter the homogeneous MPSP as studied in this dissertation has been formally defined, and it has been contrasted with the closely related BPP. A selection of the wide range of current solution techniques for these two NP-hard problems has been discussed, including simple and fast greedy heuristics, local search, branch-and-bound algorithms and more modern evolutionary and hybrid approaches.

Chapter 4

Applying ACO and local search to the homogeneous MPSP

This chapter describes how the ACO meta-heuristic was applied to the homogeneous multi-processor scheduling problem, as described in the previous chapter. The ACO methods employed here have been informed and led by the ACO approach to the BPP, AntBin, described in [Ducatelle, 2001] and [Levine and Ducatelle, 2003], but due to the different problem formulation some of the stages are rather different. The approach described here is therefore contrasted with AntBin throughout.

4.1 Defining the pheromone trail

There is not immediately as clear a definition for the pheromone trail in this problem as there was for the AS with the TSP, and so the information that will be stored in the pheromone trail must be carefully selected. As the problem is essentially to allocate jobs to processors, intuitively it seems that the trail could store the favourability of allocating a particular job to a particular processor. However because, in this version of the MPSP, all the processors will run each job at the same speed, there does not seem to be much use in storing this information - one processor will be just as suitable for a particular job as any other.

Another view of this problem that may be more useful, and the one used in AntBin,

is as a grouping problem. Because all the processors are identical, the problem can be seen as the problem of dividing the set of jobs into m subsets (remembering that m is the number of processors) such that the total running time of the jobs in each of the subsets is minimised. This approach suggests storing information about the favourability of grouping certain jobs together on a processor. Here again we have a choice, we could store the favourability of grouping certain jobs together, or we could store the favourability of storing jobs of certain running times together.

The latter approach should work well if there are several jobs of equal sizes as it allows useful information to be more efficiently stored in the pheromone trail: because the only salient information about the jobs the ants use is their running time, storing information about the job times rather than the jobs themselves allows a single pheromone trail 'entry' to cover all jobs of certain times, and not just a particular pair of jobs. If there are few jobs of equal running times, then this approach is equivalent to storing information about each job individually, and no information is lost. The use of running times was therefore the approach used, and so the pheromone value $\tau(i, j)$ encodes the favourability of allocating a jobs of running time i and running time j on the same processor. This definition is identical to the one successfully used in AntBin, where item size is equivalent to running time.

4.2 The heuristic

The heuristic information that the ants use when building their solution is also very important, it guides their search with problem specific information. However, because the ACO approach relies on multiple ants building solutions over several 'generations' the heuristic information must be quick to establish, and so only fairly simple heuristic values can be used.

As discussed in section 3.3.1.1, FFD is the best of the simple greedy heuristics for this problem. Fortunately FFD is also a very suitable algorithm to use in an ACO approach both because it provides good solutions, and also because the heuristic 'value' of each job is directly proportional to the job's running time, which is fast and simple to compute. The heuristic value used by the ants for each job j was therefore simply

its running time, giving us equation 4.1. FFD is also a good heuristic for the BPP, and so again this approach is identical to that used in AntBin.

$$\eta(j) = t_j \tag{4.1}$$

4.3 Updating the pheromone trail

To allow the ants to share information about good solutions a policy for updating the pheromone trail must be established. Dorigo's original *Ant System* followed the biological analogy closely and allowed all the ants to leave pheromone, but Stützle & Hoos have shown with their *Max-Min Ant System* (MMAS, described in detail in [Stützle and Hoos, 2000]) that allowing only the best ant, s_{best}, to leave pheromone after each iteration makes the search much more aggressive and significantly improves the performance of ACO algorithms. This was therefore the policy chosen. Equation 4.2 describes in detail the pheromone update policy used for each pair of job running times i and j in the pheromone matrix (the value $\tau(i,j)$). In this equation ρ is a parameter which defines the pheromone evaporation rate, m is the number of times jobs with running times i and j have been allocated together on a processor in s_{best}'s, solution, and $f(s_{best})$ is s_{best}'s fitness value (sec section 4.4). Essentially the pheromone matrix value for each job time pair (i,j) in s_{best}'s solution is reinforced, with pairs that occur more than once being reinforced the number of times they appear, while all other values decay according to ρ.

$$\tau(i,j) = \rho.\tau(i,j) + m.f(s_{best}) \tag{4.2}$$

Again following Stützle & Hoos' example, the best ant s_{best} can be defined as either the iteration best ant s_{ib}, or the global best ant s_{gb}, a parameter γ is used to define how often s_{ib} is used instead of s_{gb} when updating the pheromone trail. Increasing the value

of γ allows the search to be less aggressive, and encourages the ants to explore more of the solution space.

4.4 The fitness function

In order to guide the algorithms towards good solutions, a mechanism is required to assess the quality, or fitness, of a particular solution. The obvious choice would to use the inverse of the makespan of the solution. However, as is noted in [Falkenauer, 1996] (pertaining to the BPP, but also true here), there are often many different solutions which have the same makespan and so giving each an equivalent fitness value would make it impossible to distinguish promising solutions among these. Of course the makespan is a very important consideration, and so the fitness function is only intended to help distinguish between solution with equal makespans - a solution with a shorter makespan will always have a higher fitness that one with a longer makespan.

The fitness function that was implemented rewards solutions that both seem 'promising' and which are amenable to the local search described below. Every solution will have one or more 'problem' processors - those with the longest schedules (equal to the makespan of the solution) and the local search attempts to balance a solution by swapping jobs from these processors with other less-loaded processors. In order to reward an already partially well-balanced solution, and to increase the chance of finding useful swaps the fitness function rewards solutions that have well-balanced non-problem processors. This is desirable as a balanced solution is likely to be closer to the optimum, and the local search only needs to find a few swaps to reduce the problem processors' schedules and so increase the overall balance of the schedule. The function is defined in equation 4.3. In this equation opt is the optimal makespan, $ms(s)$ is the makespan of solution s, P is the set of problem processors, and A is the average schedule length of the non-problem processors.

$$f(s) = \frac{opt}{ms(s) + \sum_{j \notin P} |t(j) - A|} \tag{4.3}$$

A similar fitness function was considered which rewarded solutions that have a few

processors with comparatively short schedules, as these may seem like good candidates for the local search. However this then promotes unbalanced solutions which is not desirable.

The fitness function used is rather different to that used in AntBin, due to the differences between the BPP and the MPSP. In the BPP the aim is to use the least number of bins, and so AntBin rewards solutions which have a few near-empty bins, as it may be possible to redistribute their contents. However in the MPSP the number of processors is fixed, and so the best strategy is to try to find well-balanced solutions - near-empty processors would be inefficient.

4.5 Building a solution

Each of the ants builds a solution using a combination of the information provided by the pheromone trail, and by the heuristic function. The ants build their solution in a similar way to the FFD heuristic - they try to schedule bigger jobs earlier on a processors so that these will hold up the overall schedule the least, (i.e. they won't 'stick out' at the end of a schedule), and then they try to balance the solution by allocating smaller jobs later on. This is rather different to AntBin which tries to fill up each bin one at a time, and again this difference is a result of the fact that the number of processors is fixed, and so load-balancing them is a good strategy for the MPSP.

Every ant starts with the set of all unallocated jobs and empty processors. They firstly probabilistically pick a job to allocate using only the heuristic value $\eta(j)$. The probability of picking job j next is given in equation 4.4, in this equation β is a parameter used to define the extent to which the ants use the heuristic information (if β is 0 then no heuristic information will be used, and a job will be picked at random).

$$prob(j) = \frac{[\eta(j)]^{\beta}}{\sum_{i=1}^{n}[\eta(i)]^{\beta}} \tag{4.4}$$

Next the ants probabilistically pick a processor to which the chosen job, j_c, will be allocated using the information provided in the pheromone trail. The pheromone value between j_c and a processor p is defined in equation 4.5, in this equation δ is

a parameter which defines the pheromone value returned for processors with empty queues and $|p|$ is the number of jobs in p's queue. The equation essentially works out the total pheromone value between j_c and all the other jobs in p's queue, divided by the number of jobs in p's queue.

$$\tau_p(j_c) = \begin{cases} \frac{\sum_{i \in p} \tau(t(i), t(j_c))}{|p|} & \text{if } p\text{'s queue is not empty} \\ \delta & \text{otherwise} \end{cases} \tag{4.5}$$

The result of equation 4.5 is used to determine the probability of picking a processor p to which j_c will be allocated, this is simply defined as the normalised pheromone value between j_c and p - more pheromone means a better chance of being picked. Equation 4.6 defines this probability, in the equation α is a parameter used to define the extent to which the ants use the pheromone information (if α is 0 then no pheromone information is used), t_p is the schedule length of p, and λ is the current makespan limit.

$$prob(p) = \begin{cases} \frac{[\tau_p(j_c)]^\alpha}{\sum_{i=1}^{m} [\tau_i(j_c)]^\alpha} & \text{if } t_{j_c} + t_p \leq \lambda \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

Equation 4.6 shows that a processor p will only have a chance of being picked if j_c can fit onto p's schedule without exceeding the current makespan limit λ. λ is initially set to the makespan lower bound as given by equation 3.1, but if no processor can fit j_c onto its schedule without exceeding this value, the value of λ is incremented by one, and the entire procedure is repeated, i.e. a new job is selected and a new processor is found.

4.6 Adding local search

As discussed earlier an ACO algorithm can often be effectively improved when used in conjunction with some form of local search. The approach taken here is to allow the ants to build their solutions as described in section 4.5 above, and then the resulting solutions are taken to a local optimum by the local search mechanism. Each of these (hopefully improved) solutions is then used in the pheromone update stage. The local

search is performed on every ant, every iteration, so it needs to be fairly fast. In the case of the MPSP a simple approach is to check if any jobs could be swapped between processors which would result in a lower makespan.

However, to check exhaustively every possible swap would be computationally very intensive, and in larger problems would take an unfeasible amount of time. It also seems rather wasteful, as considering swapping jobs on two processors with schedules well below the ideal makespan would probably not contribute much to the search. It was therefore decided to only consider swapping jobs from the 'problem' processors - those with total execution times equal to the makespan of the solution, and so which are holding the solution up the most. A simple and obvious choice of technique would be to attempt a swap between each job on a problem processor with each job on any other non-problem processor.

[Falkenauer, 1996] and AntBin both use an efficient local search for the BPP which improves on this by trying more possible combinations of swaps. Following their concept, the local search used here considers one problem processor at a time and attempts to swap two jobs from the problem processor (problem jobs) with two jobs from any other (non-problem) processor in the solution (non-problem jobs), two problem jobs with one other non-problem job, and one problem job with one non-problem job. This increases the number of comparisons to be made, but tests show that this approach can find sufficiently better solutions than a simpler 'single swap' approach that the slight time increase is worth it. Clearly it would be possible to compute the best possible swap, but this would require an exhaustive search, in order to speed things up simply the first swap between a problem processor and another that will reduce the makespan of both is selected, so this is a *first ascent* local search. This is acceptable however as any swap that will reduce a problem processor's schedule will better balance the solution, and so is desirable. The search is performed on each problem processor and continues until there is no further improvement in the fitness value of the solution. An example of this algorithm in use is given in figure 4.1.

A simple tabu search was also implemented for this problem following the procedure described in [Thiesen, 1998] which uses this local search to generate candidate moves and dismisses them if the move results in a copy of a solution that has recently

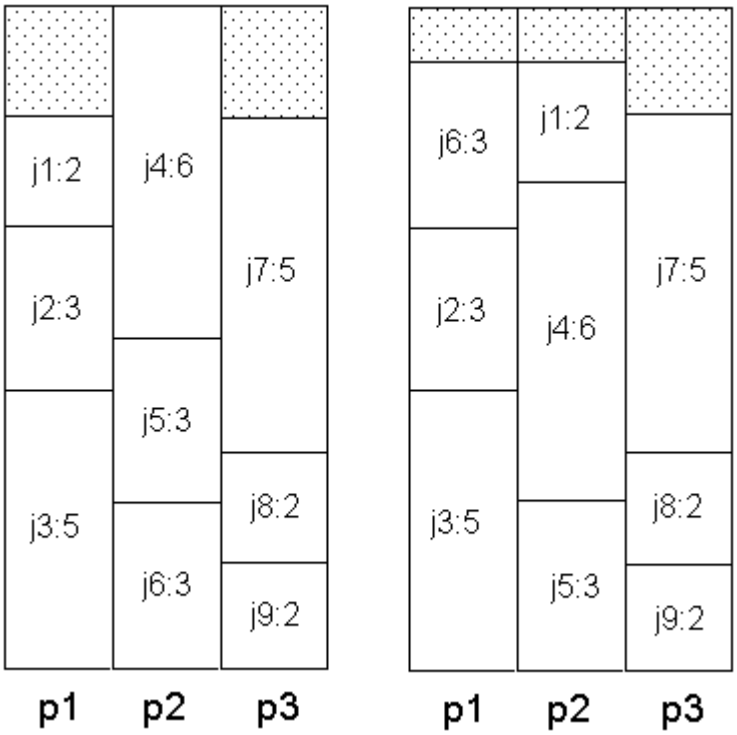

Figure 4.1: Illustration of the local search. The solution on the left hand side has a makespan of 12 and the problem processor is $p2$. The local search tries to find a job on $p2$ that can be swapped with a job on another processor which will reduce the overall makespan. The procedure finds that $j6$ can be swapped with $j1$ to reduce the overall makespan to 11, this new solution is shown on the left hand side. No further improvement is possible, and so the local search stops. Alternative moves were also possible, such as swapping jobs $j1$ and $j2$ from $p1$ with jobs $j6$ and $j5$ from $p2$.

been found. Experiments showed that this approach did not substantially improve solutions any more than a complete run of the local search and took significantly longer to run, and so it was not used.

4.7 Establishing parameter values

The parameter values used in ACO algorithms are often very important in getting good results, however the exact values are very often entirely problem dependent [Dorigo and Stützle, 2002], and cannot always be derived from features of the problem itself. Several plausible values (based on those used by other ACO algorithms in the literature, e.g. [Levine and Ducatelle, 2003]) were tested for these problem instances on several of the benchmark uniform problem instances from [Falkenauer, 1996]. Brief justifications for the values used are given below. When the values discussed were tested 'base values' for each of the parameters where used to help isolate the performance of the parameter in hand. These values were: α=20, β=20 (these two relatively high values seem to complement each other well), γ=0, δ=2, *numAnts*=10, τ_{min}=0.01, τ_0=τ_{min} and ρ=0.75.

- α determines the extent to which pheromone information is used as the ants build their solution. The lower the value, the less 'attention' the ants pay to the pheromone trail. Some ACO algorithms work well with a value of 1, i.e. the 'raw' pheromone value, but after testing values in the range 1-50 (specifically 1, 2, 5, 10, 20 and 50) it seems this algorithm works well with relatively high values (around 10-20), but not higher values because the ants then perform too little exploration.

- β, on the other hand, determines the extent to which heuristic information is used by the ants. Again, values between 1-50 were tested, and similar to α a value around 20 appeared to offer the best trade-off between following the heuristic and allowing the ants to explore.

- γ is used to indicate how many times the iteration best ant is used compared to the global best ant in pheromone updates. If γ is set to 0 then only the global

best is used, if it is set to 1 then the iteration best will be used half the time, and if it is 2 then the iteration best will be used twice for every time the global best and so on. Although this parameter would appear to help the ants explore by not 'pushing' the current global best solution at them all the time, in fact for the problem instances tested it seems that after relatively few iterations (around 2-5) the iteration best is very often largely equivalent to the global best, and so this parameter has little effect. A value of 0 was therefore used throughout.

- δ defines the pheromone value returned between any job and an empty processor. Intuition (and FFD) suggests that this value should not be too low (<1), as empty processors should be filled up fairly early on with large jobs, but a large value (>5) doesn't allow the ants to explore enough - they are effectively forced to place jobs on empty processors early on. A value of 2 worked best for most problems.

- *numAnts* defines the number of ants to use in the colony, and will clearly make a difference to the amount of search that is performed. A low value (around 1-5) speeds the algorithm up because less search is done, but also means that it takes more iterations to build up a useful pheromone trail. A high value (>20) simply slows the search down, as more ants run before each pheromone update in performed. A value of 10 appeared to be a good compromise between execution speed and time allowed to build up a useful pheromone matrix.

- τ_{min} defines the minimum pheromone value that can be returned between any particular job size and any other job size (it is the 'min' part of [Stützle and Hoos, 2000]'s Max-Min approach). This parameter is intended to stop the ants converging prematurely on a solution, and therefore should be greater than 0. A high value however means that the ants are more likely to make choices not based on pheromone information. A value between 0.01-0.03 seemed to work well.

- τ_0 is the value to which the pheromone matrix values are initialised. [Stützle and Hoos, 2000] suggest that this should be fairly high to encourage initial exploration, while the pheromone decay procedure will gradually erode

bad job pairings leaving useful information. Experimentation showed, however, that it has little effect for this problem. It was therefore set to τ_{min} for all test runs.

- ρ is the pheromone evaporation parameter and is always set to be in the range $0 \leq \rho \leq 1$. It defines how quickly the ants 'forget' past solutions. A higher value makes for a more aggressive search, in tests a value of around 0.75 gave good results.

With so many parameters, many of which seem to interact non-linearly with each other, establishing the optimal values is a very time-consuming task. The values suggested here seem plausible and work well enough, although more extensive experimentation could well improve the algorithm. It may also be that varying the parameter values during the run could be effective, such as increasing α and decreasing β as the run progresses - this could allow the ants to make more use of the heuristic information initially, but later make better use of the established pheromone information. Due to time constraints, however, such experimentation is left for future work.

4.8 Experimental results

No benchmark problem instances are available for this version of the MPSP, and so the ACO algorithm was tested on some well known BPP benchmark problem sets first published in [Falkenauer, 1996]. They include four sets of problems generated uniformly at random consisting of 120, 250, 500 and 1000 jobs respectively. Each set has 20 instances. These problems are BPP problems with known optimal numbers of bins and with a fixed bin capacity of 150. The algorithm was also tested on some different BPP problems (also from [Falkenauer, 1996]) known as 'triplets' because the optimal solution has exactly three jobs on each processor, and each processor is completely filled up to the optimal makespan of 1000. When treated as MPS problems, the number of processors available is set to the optimal number of bins (established from equation 3.2), and the goal is to find a schedule with a makespan equal to 150, or 1000 for the triplets. In some of the uniform problems the optimal makespan (as given by

equation 3.1) is lower than 150, but in order to make fair comparisons with BPP algorithms, the times presented in tables 4.1-4.4 are the time taken to find a solution with a makespan of 150, *not* the optimal makespan. Individual results for those solutions which have an optimal makespan lower than 150 given in table 4.5.

All ACO results presented are for 1000 iteration runs, and each run was performed 10 times, and the average makespan found and time taken are shown. The makespans are always integers, but if within a set of 10 runs different makespans are found the average result is presented, and so can include a decimal point. Tests were carried out on 1.6-2 GHz machines running Linux, and all programs were written in Java.

4.8.1 Comparison with other approaches

Run by run results on each of the 20 instances in each problem class are compared against the original hybrid genetic algorithm (HGGA) described in [Falkenauer, 1996], Martello and Toth's approach (MTP) described in [Martello and Toth, 1990], and the original ACO approach described in [Ducatelle, 2001] (AntBin). The ACO approach described here (AntMPS) was run on each problem 10 times, and the average time taken and makespan found for each problem set are shown in tables 4.1, 4.2, 4.3 and 4.4. Table 4.5 shows the results of AntMPS on those problems which have an optimal makespan below 150. Five instances (highlighted in bold) are not solved to the theoretical optimum makespan, but all five of these have been shown to be insoluble at this lower bound [Schoenfield, 2003]. The times presented are therefore the time taken to solve to the reachable optimum. Results are also shown, in table 4.6 and 4.7, for two classes of the triplet problems, those with 60 jobs to be scheduled onto 20 processors, and those with 120 jobs for 40 processors. 2 other sets of triplet problems with larger numbers of jobs also exist, and brief tests showed that AntMPS gave results very similar to those found on the smaller sets. The time required to run a full test suite on the larger problems is substantial, and as the results largely follow the pattern of the others, complete results are not supplied.

It is clear from the uniform results that the ACO approach described here is competitive with both HGGA and MTP. It can find optimal solutions to a few of the problems that HGGA could not solve, and is generally much faster than HGGA. MTP is

very fast on the problems it can solve optimally, but as the number of jobs increases it finds very poor solutions and takes significantly longer than AntMPS. AntBin's approach is similar to AntMPS and this is shown by the similar results, both approaches generally find an optimal or near-optimal solution, although AntBin does not perform quite as well on the larger problems. Improved results with AntBin using an iterated local search similar to the local search approach used in AntMPS (described in [Levine and Ducatelle, 2003]) are provided in the summary table 4.8 (run by run results are not provided) which show that AntBin used with extra local search gives comparable results to AntMPS. It is interesting to note that there is a strong correlation between problems which AntBin finds hard (i.e. takes a long time to solve) and those which AntMPS finds hard, again indicative of the fact that they go about the problem in a similar way.

The results for the triplet problem classes tested show rather different behaviour. AntMPS can solve only a few of the 60 job problems optimally, and cannot do so consistently, and none of the 120 jobs problems. HGGA, on the other hand, can consistently solve the majority of these. The long times taken by AntMPS for these fairly small problems are a result of the fact that the program ran for its full quota of iterations, normally the makespan shown was found early on, and the rest of the time was taken up trying to improve on this solution. As mentioned above the triplets are deliberately designed to test solution techniques, as the optimal solution has each processor filled up to the optimal makespan of 1000. AntMPS seems to find this type of problem very hard to solve optimally, but the solutions that are found generally consist of just one or two processors with a makespan of 1001, and a similar number with makespans of 999, with the rest of the processors at 1000. It is just missing the optimal solution, but still finding very good solutions, in contrast to MTP which either solves the problem optimally, or finds rather poor solutions 2 or 3 bins from the optimum. This is probably a result of the fact that the fitness function rewards well balanced solutions and so these will reinforced in the pheromone trail, and the local search is then good at balancing the load fairly evenly on ant solution. However once a solution such as the one just described is found neither the fitness function nor the local search can help to 'push' the ants towards the optimal solution, and so the problem becomes simply a

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
u120-00	48	48	15.20	48	0.10	48	1.00	150	0.26
u120-01	49	49	0.00	49	0.10	49	1.00	150	0.24
u120-02	46	46	5.80	46	29.00	46	1.00	150	0.25
u120-03	49	49	50.40	49	0.00	49	1.00	150	0.27
u120-04	50	50	0.00	50	0.00	50	1.00	150	0.24
u120-05	48	48	19.40	48	0.10	48	1.00	150	0.25
u120-06	48	48	19.00	48	0.00	48	1.00	150	0.31
u120-07	49	49	21.70	49	0.00	49	1.00	150	0.28
u120-08	50	**51**	3668.70	**51**	3681.40	50	3.00	150	1.35
u120-09	46	46	39.50	46	0.10	46	3.00	150	0.38
u120-10	52	52	0.00	52	0.10	52	1.00	150	0.26
u120-11	49	49	23.70	49	0.10	49	1.00	150	0.26
u120-12	48	48	25.70	48	0.00	48	1.00	150	0.75
u120-13	49	49	0.00	49	0.00	49	1.00	150	0.25
u120-14	50	50	0.00	50	0.00	50	1.00	150	0.35
u120-15	48	48	11.10	48	0.10	48	1.00	150	0.25
u120-16	52	52	0.00	52	0.00	52	1.00	150	0.26
u120-17	52	52	76.10	52	0.00	52	1.00	150	0.31
u120-18	49	49	14.30	49	0.00	49	1.00	150	0.26
u120-19	49	**50**	3634.70	**50**	3679.40	49	6.00	150	2.24
Averages			381.27		369.53		1.45		0.45

Table 4.1: Run by run results for the uniform problems with 120 jobs. Sub-optimal results are shown in bold type.

random search. This will occasionally be successful, but generally is not.

The hybrid improvement heuristic (HIH) described in [Alvim et al., 2002], is currently the best solution technique for these problems found in the literature, and overall comparative results with this are shown in table 4.8 (they do not provide run by run results). HIH clearly outperforms all the other approaches, finding more optimum solutions in significantly less time. HIH uses several different methods (as described in section 3.3.3) to obtain such good results, and has been finely tuned to the BPP. As an example the tabu search they employ strongly favours solutions which have bins completely filled up to the capacity. This would appear to explain their excellent re-

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
u250-00	99	99	256.70	**100**	1001.60	99	11.00	150	2.28
u250-01	100	100	47.40	100	0.20	100	1.00	150	3.51
u250-02	102	102	223.80	102	0.30	102	3.00	150	1.54
u250-03	100	100	27.00	100	0.10	100	1.00	150	1.98
u250-04	101	101	163.50	101	4.10	101	34.00	150	2.44
u250-05	101	101	477.60	**103**	522.10	**102**	308.00	150	11.81
u250-06	102	102	14.50	102	0.10	102	1.00	150	2.64
u250-07	103	**104**	6628.80	**104**	7411.80	**104**	304.00	**150.7**	827.00
u250-08	105	105	924.40	**106**	1049.10	**106**	317.00	150	17.05
u250-09	101	101	158.30	**102**	597.10	101	3.00	150	8.63
u250-10	105	105	95.60	**106**	377.20	105	2.00	150	2.01
u250-11	101	101	240.00	**102**	1075.50	101	110.00	150	3.82
u250-12	105	**106**	5996.70	**106**	6100.90	**106**	292.00	**151**	953.18
u250-13	102	**103**	6346.60	**103**	6969.20	**103**	312.00	**151**	891.43
u250-14	100	100	82.60	100	0.10	100	1.00	150	6.59
u250-15	105	105	4440.10	**106**	4672.80	**106**	305.00	150	32.46
u250-16	97	97	254.50	**98**	545.40	97	7.00	150	1.77
u250-17	100	100	38.50	100	0.10	100	1.00	150	4.29
u250-18	100	100	246.80	100	0.40	**101**	313.00	150	3.20
u250-19	102	102	68.00	102	0.10	102	1.00	150	1.31
Averages			1336.57		1516.41		116.35		138.95

Table 4.2: Run by run results for the uniform problems with 250 jobs. Sub-optimal results are indicated in bold. Note that the theoretical optimum for problem *u120-13* is 102), but the problem is in fact insoluble at this lower bound (as shown in [Gent, 1998]), and so the reachable optimum is 103 bins, or a makespan of 151.

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
u500-00	198	198	480.50	**201**	986.80	199	957.00	150	15.61
u500-01	201	201	177.70	**202**	868.50	**202**	908.00	150	20.03
u500-02	202	202	347.90	**204**	910.90	202	772.00	150	13.57
u500-03	204	204	11121.20	**206**	11412.10	**205**	1143.00	150	94.68
u500-04	206	206	267.60	**209**	844.00	206	59.00	150	12.59
u500-05	206	206	129.70	**207**	818.30	206	13.00	150	13.26
u500-06	207	207	1655.50	**210**	1854.10	**208**	890.00	150	764.44
u500-07	204	204	1834.70	**207**	2084.50	**205**	897.00	**150.7**	4451.84
u500-08	196	196	501.50	**198**	1221.80	**197**	929.00	150	25.58
u500-09	202	202	92.50	**204**	962.40	202	66.00	150	25.54
u500-10	200	200	106.20	**202**	893.60	200	3.00	150	17.08
u500-11	200	200	152.30	**202**	793.00	200	335.00	150	28.11
u500-12	199	199	1019.30	**202**	1258.20	**200**	936.00	150	33.57
u500-13	196	196	135.50	**197**	860.30	**197**	951.00	150	25.91
u500-14	204	204	951.70	**205**	1202.80	204	30.00	150	27.39
u500-15	201	201	375.20	**203**	782.90	201	26.00	150	27.43
u500-16	202	202	162.60	**204**	732.70	202	20.00	150	22.88
u500-17	198	198	336.80	**201**	754.50	198	606.00	150	27.61
u500-18	202	202	143.90	**205**	637.50	202	30.00	150	27.02
u500-19	196	196	306.80	**199**	819.20	**197**	963.00	150	26.10
Averages			1014.96		1534.91		526.70		285.01

Table 4.3: Run by run results for the uniform problems with 500 jobs. Sub-optimal results are indicated in bold.

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
u1000-00	399	399	2924.70	**403**	3279.00	**400**	3047.00	150	108.59
u1000-01	406	406	4040.20	**410**	4886.60	**408**	3017.00	150	115.82
u1000-02	411	411	6262.10	**416**	6606.10	**412**	3001.00	150	123.36
u1000-03	411	411	32714.30	**416**	40285.60	**413**	2961.00	**151**	33266.56
u1000-04	397	397	11862.00	**401**	20689.80	**399**	3073.00	150	107.83
u1000-05	399	399	3774.30	**402**	4216.30	**401**	3027.00	150	116.90
u1000-06	395	395	3033.20	**398**	3449.70	**396**	3000.00	150	109.50
u1000-07	404	404	9878.80	**406**	12674.40	**405**	3028.00	150	116.90
u1000-08	399	399	5585.20	**402**	6874.00	**401**	3006.00	150	115.13
u1000-09	397	397	8126.20	**402**	9568.20	**400**	2924.00	150	8545.13
u1000-10	400	400	3359.10	**404**	3542.80	**401**	3021.00	150	112.60
u1000-11	401	401	6782.30	**404**	7422.40	**403**	3073.00	150	114.19
u1000-12	393	393	2537.40	**396**	2714.00	**394**	3019.00	150	107.04
u1000-13	396	396	11828.80	**401**	23319.40	**398**	3111.00	150	110.73
u1000-14	394	394	5838.10	**399**	6770.90	**396**	3070.00	**151**	34279.41
u1000-15	402	402	12610.80	**407**	20458.40	**405**	3032.00	150	172.81
u1000-16	404	404	2740.80	**407**	3139.60	**405**	3010.00	150	112.50
u1000-17	404	404	2379.40	**407**	2506.40	**405**	2941.00	150	117.23
u1000-18	399	399	1329.70	**403**	1353.20	**401**	3000.00	150	111.48
u1000-19	400	400	3564.20	**405**	4109.60	**402**	3024.00	150	113.74
Averages			7058.58		9393.32		3019.25		3903.87

Table 4.4: Run by run results for the uniform problems with 1000 jobs. Sub-optimal results are indicated in bold.

problem	processors	optimum ms	ms found	time
u120-00	48	148	148	15.97
u120-01	49	148	148	0.54
u120-02	46	148	148	1.21
u120-03	49	149	149	27.4
u120-04	50	148	148	0.73
u120-05	48	149	149	1.08
u120-06	48	149	149	1.89
u120-07	49	149	149	21.77
u120-10	52	148	**149**	16.23
u120-11	49	148	**149**	25.21
u120-13	49	148	**149**	0.63
u120-14	50	148	148	1.45
u120-15	48	148	**149**	1.32
u120-16	52	148	**149**	4.12
u120-17	52	149	149	3.81
u120-18	49	149	149	1.45

Table 4.5: Results for the uniform problems with optimal makespans below 150. Five instances are not solved to the theoretical optimum (shown in bold), but these have all been shown to be insoluble at this lower bound [Schoenfield, 2003].

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
t60-00	20	20	4.00	20	9.50	**21**	37.00	**1001**	59.54
t60-01	20	20	5.80	20	12.60	**21**	42.00	**1001**	59.24
t60-02	20	20	1.50	**23**	564.20	**21**	35.00	**1000.3**	33.99
t60-03	20	20	5.90	**22**	444.70	**21**	38.00	**1000.7**	49.72
t60-04	20	20	0.60	**22**	404.60	**21**	35.00	**1000.5**	38.39
t60-05	20	20	9.00	**22**	415.20	**21**	38.00	**1001**	59.94
t60-06	20	20	284.10	**22**	485.70	**21**	38.00	**1001**	60.38
t60-07	20	**21**	295.30	**22**	395.90	**21**	37.00	**1001**	59.25
t60-08	20	20	6.80	**22**	451.60	**21**	38.00	**1000.9**	57.04
t60-09	20	20	6.20	20	9.60	**21**	39.00	**1001**	64.47
t60-10	20	20	15.30	20	0.90	**21**	40.00	**1001**	65.25
t60-11	20	20	0.60	20	6.30	**21**	34.00	**1000.1**	16.19
t60-12	20	20	2.50	20	1.50	**21**	40.00	**1001**	59.79
t60-13	20	20	4.70	**22**	385.00	**21**	36.00	**1000.7**	45.18
t60-14	20	20	5.90	**22**	400.80	**21**	38.00	**1001**	65.44
t60-15	20	20	3.40	**23**	537.40	**21**	38.00	**1001**	64.15
t60-16	20	20	2.20	**23**	528.30	**21**	34.00	**1000.2**	20.69
t60-17	20	20	9.20	**22**	429.90	**21**	37.00	**1000.8**	52.99
t60-18	20	**21**	281.10	**22**	385.60	**21**	41.00	**1000.8**	53.07
t60-19	20	20	1.60	**22**	399.50	**21**	40.00	**1000.8**	55.77
Averages			47.29		313.44		37.75		52.02

Table 4.6: Run by run results for the triplet problems with 60 jobs. Sub-optimal results are indicated in bold.

		HGGA		MTP		AntBin		AntMPS	
problem	optimum bins	bins	time	bins	time	bins	time	makespan	time
t120-00	40	40	120.90	**44**	844.30	41	135.00	**1001**	195.31
t120-01	40	40	104.00	**43**	823.00	41	131.00	**1001**	200.73
t120-02	40	40	95.90	**43**	956.40	41	141.00	**1001**	202.17
t120-03	40	40	39.10	**44**	859.30	41	188.00	**1001**	202.87
t120-04	40	40	75.80	**45**	1184.40	41	151.00	**1001**	203.39
t120-05	40	40	148.50	**45**	188.70	41	153.00	**1001**	204.08
t120-06	40	40	47.20	**45**	1054.30	41	165.00	**1001**	197.83
t120-07	40	40	61.40	**43**	777.30	41	176.00	**1001**	197.23
t120-08	40	40	36.90	**43**	642.90	41	184.00	**1001**	206.26
t120-09	40	40	255.50	**44**	1002.60	41	138.00	**1001**	200.75
t120-10	40	40	102.90	**44**	885.70	41	136.00	**1001**	195.30
t120-11	40	40	49.50	**45**	967.90	41	182.00	**1001**	201.58
t120-12	40	40	42.50	**44**	1013.50	41	136.00	**1001**	199.93
t120-13	40	40	57.30	**44**	834.80	41	161.00	**1001**	195.45
t120-14	40	40	40.90	**44**	824.30	41	134.00	**1001**	196.68
t120-15	40	40	46.80	**44**	873.00	41	133.00	**1001**	196.50
t120-16	40	40	93.00	**43**	629.30	41	169.00	**1001**	196.64
t120-17	40	40	51.10	**44**	790.20	41	139.00	**1001**	203.90
t120-18	40	40	67.30	**46**	1171.10	41	138.00	**1001**	201.78
t120-19	40	40	40.10	**45**	1075.50	41	147.00	**1001**	202.02
Averages			78.83		869.93		151.85		200.02

Table 4.7: Run by run results for the triplet problems with 120 jobs. Sub-optimal results are indicated in bold.

sults for the triplet problems. However, using such a technique for the MPSP (such as rewarding processors filled to the optimum capacity in the fitness function) would probably be over-fitting, as the optimum makespan is only a goal, and not fixed as the capacity is for the BPP.

Some criticism could also be made of the benchmark instances used here, as the way in which they are generated (generally selecting item sizes uniformly from a range less than the capacity of the bins) often means that the item sizes are fairly similar, resulting in a rather homogeneous selection. It would be interesting to see if the ACO approaches of AntMPS and AntBin could deal better with more realistic problems taken from industrial applications, for example.

Overall these results show that AntMPS cannot really compete with a specialised state of the art algorithm for the BPP. However, it can solve many test problems optimally, and it is encouraging to note that for all problems where an optimal solution is not found, the problem is solved to within one unit of the optimum. This demonstrates, I feel, the robustness of the ACO approach - that it is always capable of finding good, if occasionally sub-optimal, solutions. Also, AntMPS was designed for the MPSP, and it seems to me that some of these problems, especially the triplets, probably do not represent realistic MPS problems. It is unfortunate that no MPSP benchmark problems were found for this variation of the problem, as it would be interesting to see how it compares to other purpose-designed techniques.

AntMPS also generally requires a rather long running time even when it does find an optimal solution. This is, at least in part, due to the fact that AntMPS was designed to be fairly general so that it could easily be applied to different MPSP variations. For example, jobs are represented as individual objects which must be manipulated in comparatively complex ways, whereas BPP techniques can simply use the raw item size value as their data which makes for significantly faster computation. This does mean that techniques such as MTP and Alvim et al.'s HIH could probably not be easily re-applied to different problems. This is not a criticism of these approaches as they were solely intended to be used for the BPP, but it does explain to some extent the reason AntMPS cannot compete with them for some problems. The benefits of this generality are demonstrated in chapter 6, where some of the concepts used here are successfully

	HGGA		MTP		AntBin (upd.)		HIH		AntMPS	
class	dev.	time	dev.	time	dev.	time	dev.	time	dev.	time
u120	2	381.27	2	369.53	0	0.645	0	0.00	0	0.45
u250	2	1336.57	12	1516.41	0.8	51.51	0.2	0.20	1	138.95
u500	0	1014.96	44	1534.91	0	49.63	0	0.01	0.7	285.01
u1000	0	7058.58	78	9393.32	0	646.96	0	0.04	2	3903.87
t60	2	47.29	31	1184.40	13	n/a	0	1.11	15.8	52.02
t120	0	78.83	82	869.93	20	n/a	0	4.53	20	200.02

Table 4.8: Overall results for all problem classes tested.‘dev.’ is the total deviation from the optimum bins/makespan, and the time shown is the average execution time on a problem instance. The AntBin results shown here are from the version described in [Levine and Ducatelle, 2003] and use an iterated local search in place of the single round of local search used for the run by run results, which improves the results significantly.

	FFD		FFD + LS		AntMPS (no pher.)		AntMPS (no LS)		AntMPS	
	dev.	av. time	dev.	av. time	dev.	av. time	dev.	av. time	dev.	av. time
u120	165	0.01	14	0.06	10	146.60	146	141.92	0	0.45
u250	170	0.02	19	0.10	10	711.39	156	930.53	1.7	138.95
u500	170	0.07	20	0.42	6	3352.49	161	4881.71	0.7	285.01
u1000	185	0.46	20	2.41	18	76535.27	178	35562.60	2	3903.87

Table 4.9: Comparative results of the individual components of the ACO algorithm. The values are the total deviation from the optimal solution for all 20 instances in each problem type.

re-applied to the more general problem of scheduling jobs onto heterogeneous processors.

4.8.2 Comparing the components of the algorithm

The ACO algorithm described here is essentially a hybrid approach, and it is interesting to observe the role each of the components plays, and so this section I provide a breakdown of the results of running each of the separate components of the whole algorithm. Table 4.9 shows the results of running the FFD heuristic alone, FFD after a complete local search run, the ACO algorithm without using the pheromone trail (i.e. with an α value of 0), ACO without using local search, and finally results using the whole algorithm. Results are given as the total makespan deviation from the goal makespan of 150 over all 20 instances of each problem type (i.e. the optimum value is 3000, and the value shown is the deviation from this).

These results clearly show the extent to which the ants rely on the local search to find good solutions, the ants alone can only improve a little on the FFD heuristic. Applying local search to the FFD solution can solve some problems optimally, and even more when the ants provide different starting points without using the pheromone trail. When all the components are combined the best results are found, but the algorithm as a whole does take considerably longer to run than simply applying local search to the FFD solution, and so in applications where time is critical this might be considered the more suitable approach, as it can very quickly find near-optimal solutions (for detailed time results for the whole algorithm).

4.8.3 Analysis of an example ACO run

To illustrate the behaviour of the ACO algorithm, and the role of each of its components over a real test run, Appendix A shows the output generated by the program when run on the test problem *u120-08*. The first few lines show the optimal makespan, the makespan of FFD solution, and the makespan found by applying local search to the FFD solution.

The next lines show iterations from the ACO run. Details are displayed whenever a new global best solution is found. The first value in each line is the makespan of the solution built by an ant and the fitness value of that ant (after the ++ characters). The makespan and fitness of the solution after local search has been applied is then shown after the arrow.

The first two ant solutions (iterations 0 and 1) are actually slightly worse than the FFD makespan, but are quickly taken to their local optimum of 151 by the local search. Iterations 2-6 show steadily improving ant solutions as good job pairings are reinforced in the pheromone trail, until an ant builds a solution within 2 of the optimum without any help from the local search. Iterations 8 and 11 show the ants exploring the search space, and finding solutions which are in fact slightly worse than the best ant solution, but which the local search can use to generate even fitter solutions. The ants continue to explore without finding an improved solution until in iteration 17 an ant solution of 152 is taken to the optimum makespan by the local search.

The number of solutions with equal makespans is clear even in this short run - all of the solutions found by the local search have a makespan of 151, but all have different fitness values and so are different solutions. If the fitness function was not used no distinction could be made between these solutions and it would be very hard to guide the ants search.

The actual solution generated by the algorithm is then displayed; each line shows the total execution time and the jobs scheduled on each individual processor. Finally the time taken for the whole process and the average local search depth (the number of iterations used by local search) is shown.

Chapter 5

Heterogeneous multi-processor scheduling

This chapter presents the next problem approached in this dissertation, scheduling with heterogeneous processors. This problem essentially relaxes the rule in the previous problem that all the processors must be identical, and instead assumes that each processor can take a differing amount of time to process any given job. This version is clearly more complex than the homogeneous case, as scheduling with homogeneous processors can be seen as a special case of this problem. This chapter describes the motivation for studying this version of the MPSP, and presents the simulation model used for comparison of heterogeneous MPSP techniques proposed in [Braun et al., 2001] and used in this study. This chapter also provides an overview of current solution techniques.

5.1 Motivation

Finding good solutions for the heterogeneous MPSP is of great importance when trying to make efficient use of a heterogeneous computing system, such as a computational grid, in which there is a distributed, inter-connected suite of different machines available. The efficient scheduling of jobs on such a system is clearly critical if good use is to be made of such a valuable resource, and so the most effective scheduling algorithms

should be used. Static scheduling algorithms can be used in such a system for several different requirements [Braun et al., 2001]. The first, and most obvious, is for planning an efficient schedule for some set of jobs that are to be run at some time in the future, and to work out if sufficient time or computational resources are available to complete the run *a priori*. Static scheduling may also be useful for analysis of heterogeneous computing systems, to work out the effect that losing (or gaining) a particular piece of hardware, or some sub-network of a grid for example, will have. Static scheduling techniques can also be used to evaluate the performance of a dynamic scheduling system after it has run, to check how effectively the system is using the resources available. Finally, the static scheduling problem as defined here may in fact be applied to a dynamic scheduling problem. If the static scheduler is fast enough it could be run in 'batch mode' every few seconds to schedule jobs that have arrived on the system since the last scheduler run.

5.2 Simulation model

Real-world heterogeneous computing systems, such as computational grids, are complex combinations of different hardware, software and network components, and so it is often hard to make fair comparisons of the different techniques that are being used on various different systems. To address this problem [Braun et al., 2001] describe a benchmark simulation model for comparison of static scheduling algorithms for heterogeneous systems. They define the notion of a *metatask* as a collection of independent tasks with no inter-task dependencies, and the goal of a scheduling algorithm is to minimise the total execution time of the metatask. As the scheduling is performed statically all necessary information about the jobs in the metatask and processors in the system (to return to my original terminology) is assumed to be available *a priori*. Essentially, the expected running time of each job on each processor must be known, and this information can be stored in an 'expected time to compute' (ETC) matrix. A row in an ETC matrix contains the ETC for a single job on each of the available processors, and so any ETC matrix will have $n \times m$ entries, where n is the number of jobs and m is the number of processors. A simple example ETC matrix with details for 5

	processor 1	processor 2	processor 3
job 1	25	34	17
job 2	13	25	8
job 3	45	69	33
job 4	7	11	6
job 5	19	29	13

Table 5.1: An example ETC matrix. The figures indicate the time that processor *m* is expected to take to execute job *n*. This is a *consistent* ETC matrix

jobs and 3 processors is given in table 5.1.

In any real heterogeneous computing system the running time of a particular job is not the only factor that must be taken into consideration when allocating jobs, the time that it takes to move the executables and data associated with each job should also be considered. To resolve this the entries in the ETC matrix are assumed to include such overheads. Also, if a job is not executable on a particular processor (for whatever reason) then the entry in the ETC matrix is set to infinity.

In order to simulate various possible heterogeneous scheduling problems as realistically as possible [Braun et al., 2001] define different types of ETC matrix according to three metrics: *task heterogeneity*, *machine heterogeneity* and *consistency*. The task heterogeneity is defined as the amount of variance possible among the execution times of the jobs, two possible values were defined: *high* and *low*. Machine heterogeneity, on the other hand, represents the possible variation of the running time of a particular job across all the processors, and again has two values: *high* and *low*. In order to try to capture some other possible features of real scheduling problems, three different ETC consistencies were used: *consistent*, *inconsistent* and *semi-consistent*. An ETC matrix is said to be consistent if whenever a processor p_j executes a job j_i faster than another processor p_k, then p_j will execute all other jobs faster than p_k. A consistent ETC matrix can therefore be seen as modelling a heterogeneous system in which the processors differ only in their processing speed. In an inconsistent ETC a processor p_j may execute some jobs faster then p_k and some slower. An inconsistent ETC matrix could therefore simulate a network in which there are different types of machine available,

e.g. a UNIX machine may perform jobs that involve a lot of symbolic computation faster than a Windows machine, but will perform jobs that involve a lot of floating point arithmetic slower. A semi-consistent ETC matrix is an inconsistent matrix which has a consistent sub-matrix of a predefined size, and so could simulate, for example, a computational grid which incorporates a sub-network of similar UNIX machines (but with different processor speeds), but also includes an array of different computational devices.

These different considerations combine to leave us with 12 distinct types of possible ETC matrix (e.g. high task, low machine heterogeneity in an inconsistent matrix, etc.) which simulate a range of different possible heterogeneous systems. The matrices used in the comparison study of [Braun et al., 2001] were randomly generated with various constraints to attempt to simulate each of the matrix types described above as realistically as possible. The methods used to generate the matrices are briefly described here. Initially a $m \times 1$ 'baseline' vector B is generated by repeatedly selecting m uniform random floating point values from between 1 and ϕ_b, the upper bound on values in B. Then the ETC matrix is constructed by taking each value $B(i)$ in B and multiplying it by a uniform random number $x_r^{i,k}$ which has an upper bound of ϕ_r. $x_r^{i,k}$ is known as a *row multiplier*. Each row in the ETC matrix is then given by $ETC(j_i, p_k) = B(i) \times x_r^{i,k}$ for $0 \leq k \leq n$. The vector B is not used in the actual matrix. This process is repeated for each row until the $m \times n$ matrix is full. Each of the different task and machine heterogeneities described above is modelled by using different baseline values: high task heterogeneity was represented by setting ϕ_b=3000 and low task heterogeneity used ϕ_b=100. High machine heterogeneity was represented by setting ϕ_r=1000, and low machine heterogeneity was modelled using ϕ_r=10. To model a consistent matrix each row in the matrix was sorted independently, with processor p_1 always being the fastest, and p_m being the slowest. Inconsistent matrices were not sorted at all and are left in the random state in which they are generated. Semi-consistent matrices are generated by extracting the row elements $\{0, 2, 4, \ldots\}$ of each row i, sorting them and then replacing them in order, while the elements $\{1, 3, 5, \ldots\}$ are left in their original order, this means that the even columns are consistent while the odd columns are (generally) inconsistent.

For their study 100 matrices were generated of each of the 12 possible types, modelling 16 processors and 512 jobs for all matrices. Exactly the same matrices used their study were used to test the ACO approach described in the next chapter.

5.3 Current solution techniques

[Braun et al., 2001] compare 11 heuristics for this problem, and the reader is referred to the paper for details of each approach. Some of the significant ones are briefly discussed below, along with techniques employed by other researchers on similar problems.

5.3.1 Greedy heuristics

As with the previous problem, simple greedy heuristics are a common approach. One of the simplest techniques is known as 'opportunistic load balancing' (OLB) which simply schedules each job, in arbitrary order, to the next machine that is expected to be available. The intuition is simply to try and keep the processors balanced, but because the algorithm doesn't take into account expected task execution times it finds rather poor solutions. An alternative is therefore simply to schedule each job on the processor on which it will be executed the fastest, this known as the minimum execution time (MET) heuristic, but in a consistent matrix where one processor will run all jobs faster than all the others MET will clearly not balance the load across the processors well.

A more effective approach is to assign each job, again in arbitrary order, to the processor on which it is expected to finish earliest. The completion time *ct* that a processor p can 'offer' a job j established by working out the total execution time of any existing jobs on the p's queue (written as $j_i \in p$) and then adding this to the ETC of j on p, as shown in equations 5.1, where $|p|$ is the number of jobs on a processor p's queue and $ETC(j,p)$ is the ETC of j on p taken from the ETC matrix.

$$ct(j,p) = ETC(j,p) + \sum_{i=1}^{|p|} ETC(j_i,p) \forall j_i \in p \tag{5.1}$$

The completion time is therefore a dynamic value which will change as jobs are

allocated to processors throughout the solution building process, and so is more useful than just using a raw ETC matrix value. Simply allocating each job to the processor that offers it the lowest *ct* value is known as the 'minimum completion time' (MCT) heuristic. The MCT approach therefore combines the benefits of both OLB and MET, whilst avoiding some of their pitfalls.

The best of the simple heuristics, Min-min, is also based on the idea of job completion times, but takes it a step further. Min-min starts with the set of all unallocated tasks, and the MCT value for each job, and the associated processor p^j_{best}, is computed. Finally the job with overall minimum MCT (hence min-min) is selected and allocated to the corresponding processor. This process is repeated until all jobs are allocated. Min-min is an effective approach because each job has a good chance of being allocated to a suitable processor, and also because it minimises the effect allocating each job will have on increasing the overall makespan, and therefore balances the load to an extent. Min-min is clearly more complex than MCT as it has to check the MCT for every job every iteration, but it does consistently find better solutions than MCT.

[Wu and Shu, 2001] describe a rather more complex greedy search, known as the 'Relative Cost' (RC) heuristic which is intended to improve on the Min-min approach. They argue that there are two important considerations when allocating a job to a processor on a heterogeneous computing system: (1) matching - a job should be allocated to a processor that will run it fastest, and (2) load balancing - the load should be balanced over all processors to ensure the whole system is being used as efficiently as possible. RC therefore uses two cost measures intended to address both of these to compute the best job to schedule next and the processor to which it should be scheduled: the *static relative cost* (SRC) and the *dynamic relative cost* (DRC). The SRC of a job processor pair (j_n, p_m) is simply the ETC of j_n on p_m divided by the average ETC of j_n on all processors, as shown in 5.2, and is fixed for the entire run. The DRC is re-calculated after each job is scheduled, and is the completion time, *ct*, of each job processor pair j_n on p_m, divided by the average *ct* of j_n on all processors, shown in equation 5.3.

$$SRC(j_n, p_m) = \frac{ETC(j_n, p_m)}{\sum_{i=1}^{m} ETC(j_n, p_i)/m} \tag{5.2}$$

$$DRC(j_n, p_m) = \frac{ct(j_n, p_m)}{\sum_{i=1}^{m} ct(j_n, p_i)/m} \tag{5.3}$$

The DRC and SRC are calculated for each unscheduled job every iteration (i.e. after each job is scheduled), and the job j_{best} with the overall minimum $SRC(j_n, p_m)^{\alpha} \times DRC(j_n, p_m)$ (where α is a parameter used to control the effect of the SRC, and is set to 0.5 throughout their experiments) value is scheduled next onto the corresponding processor p_{best}. The SRC is effectively acting as the matching criterion, and the DRC is acting as the load balancing criterion. The priority of a job in this scheme is no longer tied (as it is with Min-min) to the job running time, but to a relative value. The hope is that RC will therefore balance the load on the processors better than Min-min, while still allocating each job to its best processor. Experiments show that RC can indeed find better solutions than Min-min for some problems.

5.3.2 Tabu search

[Braun et al., 2001] also describe a tabu search (TS) approach to this problem. The approach described starts with a randomly generated initial solution. All possible swaps of any two jobs between all processors is then considered, the best swap is selected and the two jobs are reassigned to the corresponding processors. This local search is repeated until there is no further improvement. This, locally optimised, solution is then added to a tabu list to ensure that this region of the search space will not be needlessly visited again. The algorithm then makes a 'long hop' to another region of the search space by using another randomly generated solution that must differ from any solution in the tabu list by at least half the processor assignments. This solution is then locally optimised, and the entire process is repeated until the maximum number of iterations is met.

This is a simple TS strategy that is effectively performing random re-start hill-climbing (see section 2.3 for a more detailed explanation), but avoiding previously visited regions of the search space using the tabu list.

5.3.3 Evolutionary approaches

The best solution technique found in [Braun et al., 2001]'s comparison was a genetic algorithm (GA). The GA described works on *chromosomes* which represent a complete solution to the problem. Each chromosome is simply a array of n elements, in which position i represents job i, and each entry in the array is a value between 1 and m which represents the processor to which the corresponding job is allocated. The main steps of the algorithm are described below.

1. Generate an initial population of 200 chromosomes. Two policies were used, either use 200 randomly generated chromosomes, or use 199 randomly generated ones, plus the Min-min solution (known as *seeding* the population).

2. Evaluate the 'fitness' of each individual. The fitness is defined simply as the makespan of the solution encoded by a chromosome, a lower fitness is therefore preferable.

3. Create the next generation using:

   - Selection of the fitter individuals, a rank-based roulette wheel scheme was used (see [Mitchell, 1998] for details) which duplicated individuals with a probability according to their fitness. An *elitist* strategy was also employed which guarantees that the fittest individual is always duplicated in the next generation.
   - Crossover between random pairs of individuals. Single point crossover was used and each chromosome was considered for crossover with a probability of 60%.
   - Random mutation of individuals. A chromosome is randomly selected, then a random task in the chromosome is randomly assigned to a new processor. Every chromosome is considered for mutation with a probability of 40%.

4. While the stopping criteria are not met, repeat from step 2. The GA stops when either 1000 iterations is completed, there has been no change in the elite chromosome for 150 iterations, or all chromosomes have converged to the same solution.

This GA finds the best or equal best solutions to all the ETC matrix types tested in [Braun et al., 2001], although it does takes significantly longer that Min-min which was the second best technique for most problems (around 60 seconds compared to under a second for Min-Min).

[Abraham et al., 2000] also propose a GA for scheduling independent jobs on a computational grid, and describe two effective hybrid algorithms which combine the GA with simulated annealing (SA) and a TS. Their GA population is initialised with solutions built by a heuristic known as 'longest job on the fastest resource' (LJFR). The chromosomes encode a complete schedule and then standard GA operators are applied to try to improve the solution. The hybrid GA-TS algorithm simply replaces the genetic mutation operator with a tabu search, and the GA-SA hybrid essentially performs a SA algorithm on the resulting offspring chromosomes after each new generation is formed. Unfortunately [Abraham et al., 2000] do not supply any performance results for their algorithms, and so direct comparison with other techniques cannot be made.

5.4 Summary

This chapter has provided motivational reasons for studying the heterogeneous MPSP as a problem which has several important applications, a good example of which is the efficient allocation of resources in a computational grid. The concept of an ETC matrix, as proposed in [Braun et al., 2001], has been presented as a useful benchmark simulation model for comparison of approaches to this problem, and the way in which ETC matrices can be generated to model several possible heterogeneous computing environments has been described. This chapter has also introduced several current techniques for scheduling in a heterogeneous computing environment, which represent a spectrum of techniques ranging from fast heuristics to slower, but effective, evolutionary approaches.

Chapter 6

Applying ACO and local search to the heterogeneous MPSP

This chapter describes how the ACO meta-heuristic and local search techniques can be adapted to attack the heterogeneous MPSP as defined in the previous chapter.

6.1 Defining the pheromone trail

The fact that jobs will run at different speeds on different processors means that this problem cannot be approached as the same sort of grouping problem as for the homogeneous case. However we can exploit this fact to use what is perhaps the more intuitive pheromone trail definition, that certain jobs may have certain 'affinities' with certain processors, and so it would be useful to store information about good processors for each job. The pheromone value $\tau(i, j)$ was therefore selected to represent the favourability of scheduling a particular job i onto a particular processor j. The pheromone matrix will thus have a single entry for each job-processor pair in the problem.

6.2 The heuristic

As discussed earlier [Braun et al., 2001] show that the Min-min heuristic is a very effective algorithm for this problem (it was only consistently beaten by the GA ap-

proach). Min-min suggests that the heuristic value of particular job should be proportional to the minimum completion time (MCT) of the job, that is the time a job can be expected to finish on j's 'best' processor p^j_{best}. This processor is established for each job according to equation 6.1. In this equation $t(p_i)$ is the current total running time of a processor p_i, and $ETC(j,p)$ is the ETC matrix entry representing the ETC of j on p.

$$min_{1\leq i\leq m}(t_{p_i} + ETC(j,p_i)) \tag{6.1}$$

The completion time of job j on p^j_{best}, i.e. j's minimum completion time, is then used for the heuristic function, a lower value is preferable and so the inverse is used. The resulting $\eta(j)$ function use by the ants is defined in equation 6.2.

$$\eta(j) = \frac{1}{ct(j,p^j_{best})} \tag{6.2}$$

If the minimum completion time of j is large, as it often is when dealing with the values in the ETC matrices, $\eta(j)$ will be a very small value. To allow this value to be effectively controlled with the β parameter, it is necessary to 'scale' the heuristic value up. Therefore in the implementation of this function all the $\eta(j)$ values are computed for each job and then the job list is sorted into descending order of these values. The value then used in equation 6.5 below is the jobs position in the sorted list, which will be in the range 1 - n. A job with a lower $\eta(j)$ value will have thus have a relatively larger integer heuristic value.

6.3 The fitness function

The goal of the fitness function is essentially to help the algorithm discern between high and low quality solutions. In the homogeneous case this was important as several different solutions could have equal makespans. In this problem the chances of two different solutions having equal makespans is very low, due to the more complex nature

of the problem. It was therefore decided that the fitness function for this problem could simply be the inverse of the makespan of the solution, as stated in equation 6.3.

$$f(s) = \frac{1}{ms(s)} \tag{6.3}$$

6.4 Updating the pheromone trail

The pheromone trail is updated in a very similar way to the previous problem, and the MMAS approach is used again. As mentioned above the pheromone matrix will have an entry for each job-processor pair in the problem, and each such pair in s_{best}'s solution is reinforced in proportion to the relative fitness value of s_{best} compared to the global best solution s_{gb}, if γ is 0 and so only s_{gb} is used for updates this will always be 1 (s_{gb} is initially set to the Min-min solution). Equation 6.4 defines the policy.

$$\tau(i,j) = \begin{cases} \rho.\tau(i,j) + \frac{f(s_{best})}{f(s_{gb})} & \text{if job } i \text{ is allocated to processor } j \text{ in } s_{best} \\ \rho.\tau(i,j) & \text{otherwise} \end{cases} \tag{6.4}$$

6.5 Building a solution

There are a few possible solution building schemes which could be used for this problem and some of the alternatives are discussed here. A simple strategy, following the MET approach, would be to allocate each job j, in arbitrary order, to a processor p picked probabilistically with respect to the pheromone value between j and p, and the ETC of j on p (a lower value is preferable). However the ETC is a static value, and so as with MET, this will not balance the load on the processors well.

The completion time of j on p seems like a more sensible metric to use as it both picks a good processor for each job and also takes into account the current load on a processor. This suggests another possible approach: each job j is allocated, again

in arbitrary order, to a processor p which is selected probabilistically with respect to the pheromone between j and p, and the completion time of j on p, $ct(j,p)$ (again, a lower value is preferable). This approach can be thus viewed as following the concept of the MCT heuristic.

The solution building technique that was in fact used for this ACO approach is an attempt to follow the concept of the best heuristic method, Min-min. Firstly the processor p^j_{best} which will complete each job j earliest is established (following the process described in section 6.2 above). A job j is then picked to schedule next based on the pheromone value between j and its best processor p^j_{best}, and j's heuristic value (as determined by the process in section 6.2 above). The probability of selecting job j to schedule next is given by equation 6.5.

$$prob(j) = \frac{[\tau(j, p^j_{best})]^{\alpha}.[\eta(j)]^{\beta}}{\sum_{i=1}^{n}[\tau(i, p^i_{best})]^{\alpha}.[\eta(i)]^{\beta}} \tag{6.5}$$

A job is then selected based in this value, and the chosen job j_c is then allocated to $p^{j_c}_{best}$. This process is repeated until all jobs have been scheduled and a complete solution has been built. The pheromone trail update procedure is then used on the iteration best ant.

This scheme does perhaps give the ants less flexibility than the other approaches proposed, as a job can only ever be allocated to its best processor. However, because the MCT of each job is re-established after each allocation, taking the new processor loads into account, the best processor for each job can change as the solution is built. This gives the ants enough flexibility to explore, whilst also ensuring that each job is matched with a good processor. This approach proved to be effective (as can be seen in section 6.9 below), but further experimentation with the alternative approaches, especially the MCT-type approach, would be interesting and could possibly improve on this scheme.

It was observed in test runs that the ants often take some time to start building good solutions because it takes a few iterations to before the pheromone trail is populated with good job-processor pairings. To attempt to resolve this a pheromone *seeding* strategy was used which initially sets the global best solution s_{gb} to be the Min-min

solution after local and tabu searches (described below), and a pheromone update is performed once before the ants start building solutions. This seems to work well as the ants start producing solutions better than or near the global best almost immediately (results are provided in section 6.9 below).

6.6 Adding local search

A similar local search to that used for the homogeneous problem was implemented for this problem. The fact that jobs may take different times on different processors does not really affect the approach. When considering swapping two jobs j_1, currently allocated to p_a, and j_2, currently allocated to p_b the procedure must of course consider the ETC of j_i on p_2, and j_2 on p_1, but other than this the search can take exactly the same approach.

A local search algorithm identical to the approach used in the previous problem was implemented, along with several other plausible techniques. Experimentation showed that for this problem a local search which only considered swaps of individual pairs of jobs was considerably faster than comparing 2 by 2, 2 by 1 and 1 by 1 as was done previously, and the solutions found by the previous approach were not sufficiently better to warrant the extra time. However, if only single pairs of jobs are considered it becomes feasible to search through all possible swaps and to select the best, i.e. the swap that reduces the makespan of two processors the most, and not just the first swap that improves the solution.

In this problem tests showed that performing an exhaustive search on single swaps actually proved to find better solutions faster than a 'first-improvement' search over a greater number of possible swaps. Also in this problem, as a result of the fact that jobs can take very different times on different processors, it is useful to search for possible 'transfers' of jobs where a job is simply moved from one processor to another. As in the previous local search in order to speed up the search, and to avoid unnecessary comparisons, only swaps and transfers from a problem processor to the other (non-problem) processors are considered.

The local search procedure implemented for this problem, therefore, searches

	p1	p2
j1	3	4
j2	4	3
j3	2	4
j4	3	1

Table 6.1: The ETC matrix for the local search example in figure 6.1.

through all possible transfers and single job pair swaps from the first problem processor in the solution and all other non-problem processors in the solution, and then performs the move that improves the makespan of both processors the most. This procedure is repeated until no further improvement is possible. An example of this procedure is shown in figure 6.1, the ETC matrix for the problem is given in table 6.1. When used in conjunction with the ACO algorithm, each solution found by an ant is optimised with this search, and this improved solution is used n the pheromone update. Individual results of applying this search to the Min-min solution are also provided in section 6.9.

6.7 Tabu search

A TS mechanism was implemented for this problem which largely follows [Thiesen, 1998]'s approach described in detail earlier in section 3.3.1.2, but which uses the local search procedure just described to generate new solutions. The solution generation method is the same as for the local search, but swaps or transfers which would lead to a solution which is identical to one already on the tabu list are not allowed. New solutions are added to the tabu list for a set number of iterations, the value that gave good results for Thiesen, around 10, also worked well here.

Storing complete information about solutions on the tabu list would be computationally intensive (and therefore time-consuming) for larger problems, and as this must be done several times per iteration would detract from the feasibility of using a TS as part of an ACO algorithm. In order to resolve this, as suggested by [Thiesen, 1998], a single *hash code* is computed for each solution which reduces the relevant information

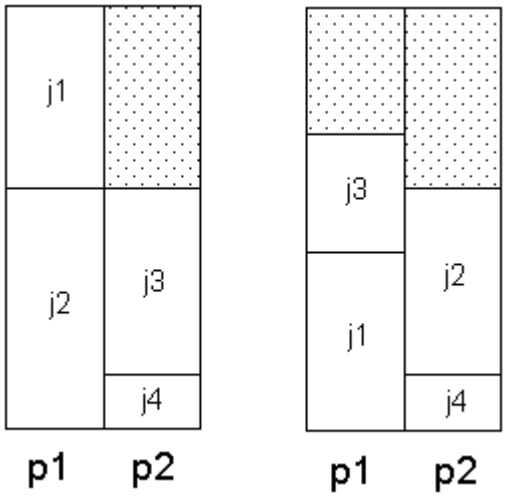

Figure 6.1: Illustration of the local search for heterogeneous processors. The solution on the left hand side has a makespan of 7 and the problem processor is $p1$. The local search tries to find a job on $p1$ that can be swapped with a job on another processor which will reduce the overall makespan. The procedure finds that $j2$ can be swapped with $j3$ to reduce the overall makespan to 5, this new solution is shown on the left hand side.

required from a solution to a single number. The hash coding scheme is illustrated in the following equations. Firstly the 'sum-squared' value of each processor p in the solution is established in equation 6.6, where $|p|$ is the number of jobs allocated to p and j_{pi} is the ID number of the ith job assigned to p. The hash code of the solution s, $hc(s)$ is then established using equation 6.7.

$$sumsq(p) = \sum_{i=1}^{|p|} j_{pi}^2 \tag{6.6}$$

$$hc(s) = \sum_{i=1}^{m} i \times sumsq(p_i) \tag{6.7}$$

This method does not guarantee that different solutions will never hash to the same code and so a solution could be erroneously rejected as 'tabu', but [Thiesen, 1998] notes that this possibility is very small. Occasionally rejecting a solution mistakenly will not affect the performance of the search significantly, and using a conflict management system to resolve this would add disproportionately to the execution time.

TS approaches also generally include some means of generating new solutions if no improvement has occurred for some time. The technique employed here is simply to swap some number, set to 20, of random job pairs. This is performed after the solution has not improved for 100 iterations.

This TS is fairly simple but does work well on the problem instances used here. Obviously the number of iterations it is allowed to run for affects its performance, but observation of its behaviour shows that it can generally quickly improve on the solutions found by local search, but tends to get 'stuck' after an initial run of around 100-1000 iterations. Allowing it to run for a much larger number of iterations can often improve on this, but not by a great deal. Experimental results for the TS applied to the Min-min solution are given in section 6.9 below.

When the TS is used in conjunction with the ACO algorithm the TS is simply used for *nTrials* (a parameter) iterations to try and improve the solution of the iteration best ant (which will already have had local search applied to it). The TS is not applied to

every ant as for the local search due to the longer running time. As can be seen in the results below it cannot always improve on the locally optimised ant solution, but it can sometimes 'break through' local optima, and adds to the performance of the algorithm as a whole significantly.

6.8 Establishing parameter values

As for the first problem we have a large number of parameter values to test and set. 12 different ETC matrix types were used as benchmarks for this approach (see section 6.9 below for details) and so one instance from each type were used to test different parameter values. A brief rationale for each value used is given below.

- α determines the extent to which pheromone information is used as the ants build their solution. Pheromone is critical for the success of this algorithm, and having tested values between 1-50, it seems this algorithm works best with a relatively high value of 10 for all problems.

- β determines the extent to which heuristic information is used by the ants. Again, values between 1-50 were tested, and a value of 10 worked well for most problem types, and to allow fair comparison this was the value used for all results presented here. Further tests on longer runs show that lower values produce better long term results for the inconsistent matrices. As a longer run progresses the ants' solutions become significantly better than that produced by the Min-min heuristic, and it was observed that in long runs the ants could not reproduce solutions as close to the global best as well as they could earlier on in the run. It was therefore hypothesised that the high β value which is necessary to get good solutions to begin with might actually be limiting the ants later in the run. A β decay mechanism was therefore implemented to allow this value to gradually decrease as the run progresses. Tests showed, however, that as the β value decays the ants start producing worse solutions, and so this feature was not used.

- γ is used to indicate how many times the iteration best ant is used in place of the global best ant in pheromone updates. Initially it seemed that a value of 0, as

used previously, worked best, and for comparison this was the value used in the results shown below. However, again, longer test runs show that higher values could work well for some problem instances. Specifically, the very best result found for the problem *u-c-hihi.0* used a γ value of 1.

- *numAnts* defines the number of ants to use in the colony. Similar to the previous a problem a value of 10 was the best compromise between amount of search per iteration and speed of execution.

- τ_{min} defines the minimum pheromone value that can be returned between any job and any processor. A value of 0.01 again worked well, balancing exploration and avoiding bad job-processor pairings.

- τ_0 is the value to which the pheromone matrix values are initialised, it was set once again to τ_{min} for all test runs.

- ρ is the pheromone evaporation parameter, a value of 0.75 as used previously (and in other ACO algorithms) gives good results.

- *nTrials* is the number of trials performed in the tabu search phase. As the results below show, a value of around 1000 allows the tabu search to help improve solutions enough, while a longer run would slow execution time without providing significantly better results.

Once again due to the time taken for a decent sized run of the whole ACO algorithm, and due to the inbuilt stochasticity of the approach finding the optimal values for these parameters is a complex and very time-consuming task. The values used work well enough, as the results below show, but there is undoubtedly room for improvement. The β decay mechanism did not prove to be helpful, but it may be that changing the values of other parameters, γ in particular, over the course of a run would improve results. There is a lot of scope for future work to experiment with changing these values for various considerations, such as which work best for particular problem types, or perhaps different values might work well depending on the length of run desired: a high β value may provide good solutions quickly, but a lower value may provide better results after a longer period of time.

6.9 Experimental results

[Braun et al., 2001] provide a comparison of 11 heuristics for scheduling on heterogeneous processor using ETC matrices as their simulation models, the problem sets that they used are used as benchmarks for testing the approaches described above. The data set includes 12 different ETC matrix types, each of which has 100 test instances. The different instances are identified according to the following scheme: *w-x-yyzz.n*, where:

- *w* denotes the probability distribution used; only uniform distributions were used so this is *u* for all files.
- *x* denotes the type of consistency, one of:
  - *c*: consistent matrix
  - *i*: inconsistent matrix
  - *s*: semi-consistent
- *yy* denotes the task heterogeneity, one of:
  - *hi*: high heterogeneity
  - *lo*: low heterogeneity
- *zz* denotes machine heterogeneity, one of
  - *hi*: high heterogeneity
  - *lo*: low heterogeneity
- *n* is the test case number, numbered from 0 to 99.

A description of the differences in each of these and an explanation of how they were generated is provided in section 5.2 of this dissertation.

6.9.1 Comparison with other approaches

Table 6.2 shows the results of the Min-min heuristic and the GA from [Braun et al., 2001], and the results of running my re-implementation of the RC heuristic from [Wu and Shu, 2001] on all 1200 of the problem instances.[1] These results are compared with the makespans found by applying the local search described above to the Min-min solution, and by applying the tabu search described above to the Min-min solution. Figure 6.2 compares the normalised makespans found (with the Min-min solution set to 1) of the different techniques. For these tests the tabu list size was always set to 10, and the tabu search was allowed to run for 1000 iterations. Tests were carried out on 1.6-2 GHz machines running Linux, and all programs were written in Java.

It is interesting to note that simply applying the local search to the Min-min solution consistently finds a shorter makespan than the GA, and in significantly less time - on average around 0.3 seconds, compared to 60 seconds for the GA. The tabu searches can improve on this solution, and still take only around 10 seconds. However comparison of times may not be entirely fair as the GA program was not available for testing on similar machines, and as the paper presenting it was published in 2001 it may be that it would run faster on more modern machines.

I have also successfully applied the local and tabu searches to the solution found by the RC heuristic, and tests show that this combination is very effective for the inconsistent matrices (the *u-i* problems) and finds better results that when Min-min is used to provide an initial solution. This combination did, however, find noticeably worse results for the other problem types. Nonetheless, I think that this demonstrates that the local and tabu searches described here can be effectively used to improve solutions found by other means.

The ACO approach as a whole takes a comparatively long time to build solutions, approximately 12 seconds per iteration (when 10 ants are used), and so it would be unfeasible to run even a fair-sized test on all 1200 problems. Results are therefore only provided for the first problem (where n=0) in each class of ETC matrix. The actual

[1] The results provided in [Wu and Shu, 2001] show that the RC heuristic can find better solutions than the GA described in [Braun et al., 2001] on their problem instances (which were generated in a similar way to the ones used here, but are not identical). It should be noted that the RC results shown here are from *my* re-implementation of the RC heuristic tested on the instances used in [Braun et al., 2001].

	Min-min		RC		GA		Min-min+LS		Min-min+Tabu	
type	makespan	time	makespan	time	makespan	time	makespan	time	makespan	time
u-c-hihi	8428258.43	0.17	9379636.66	0.96	7906149.42	63.88	7667606.81	0.49	**7649349.87**	14.06
u-c-hilo	162745.18	0.17	162262.35	1.01	155604.90	64.65	153990.94	0.39	**153924.38**	10.97
u-c-lohi	283083.40	0.16	315314.19	0.98	266723.49	62.99	258656.25	0.49	**258147.76**	13.98
u-c-lolo	5460.25	0.15	5436.06	0.90	5221.13	65.01	5167.83	0.38	**5164.88**	10.94
u-i-hihi	3632360.64	0.22	3564526.84	0.99	3206790.90	68.93	3053883.41	0.32	**3053575.69**	9.89
u-i-hilo	82413.30	0.24	80643.13	1.00	76969.72	66.70	**75594.93**	0.33	**75594.93**	9.98
u-i-lohi	122044.94	0.24	118939.07	0.98	108747.50	68.54	103355.06	0.32	**103341.11**	9.89
u-i-lolo	2777.16	0.25	2712.11	0.95	2599.61	66.07	**2552.42**	0.32	**2552.42**	9.87
u-s-hihi	4897763.92	0.19	4826409.95	0.91	4436362.83	64.73	4255904.34	0.34	**4254347.37**	10.81
u-s-hilo	105157.39	0.18	101571.52	0.98	99117.78	64.25	97587.99	0.33	**97581.37**	10.28
u-s-lohi	163927.91	0.16	161757.31	0.98	149018.93	63.17	142941.47	0.35	**142915.13**	10.76
u-s-lolo	3527.45	0.17	3411.62	0.90	3323.14	63.03	3275.20	0.33	**3275.13**	10.27

Table 6.2: Results of complete runs over all problem instances. The best result for each problem type is indicated in bold.

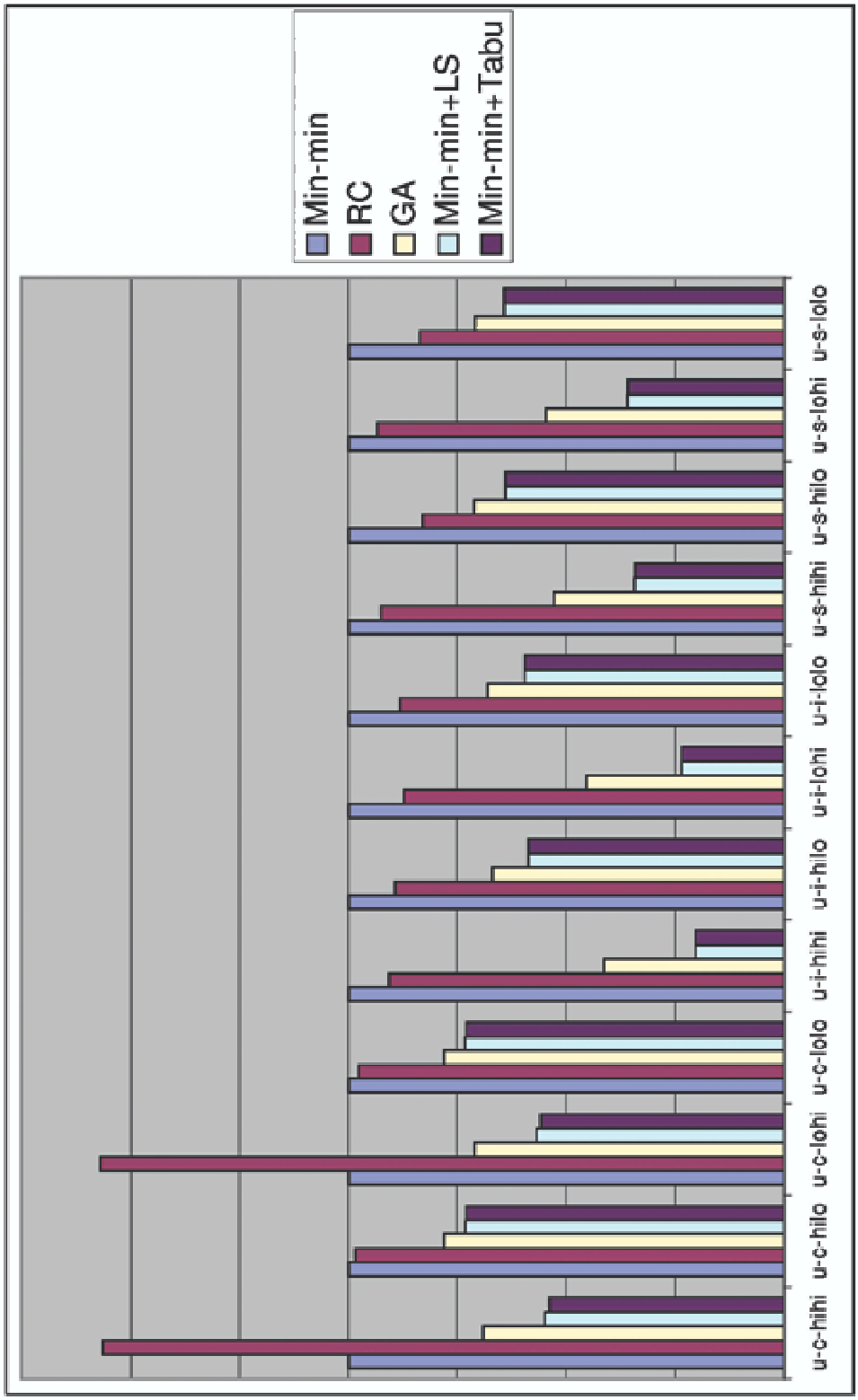

Figure 6.2: Normalised full results (with the Min-min makespan set to 1) for all approaches except the ACO algorithm. Note that the Y axis starts at 0.8.

makespans found are provided in table 6.3, along with the results from the other approaches for the same problem. Figure 6.3 compares the normalised makespan found by each approach.

For these tests the ACO algorithm was allowed to run for 1000 iterations which took an average of 12792 seconds (just over 3.5 hours). The ACO algorithm is allowed to run for so long because this allows it reasonable time to build up a useful pheromone trail. The ants need a decent length run to find solutions which significantly improve on the other solutions. To allow fair comparison the tabu searches were run for 1,000,000 iterations which took 12967 seconds for the Min-min+Tabu search. The Min-min search took an average of 0.19 seconds, the Min-min+LS algorithm ran for an average of 0.37 seconds, and the GA took an average of 65.16 seconds. It would perhaps have been fairer to allow the GA to run for an equivalent amount of time to the ACO algorithm but the program was not available for testing. However, [Braun et al., 2001] note that the GA was usually stopped because the elite chromosome (best solution) had not changed for 150 iterations, so it may be that GA would not have found better solutions with much more time anyway.

From these results it is clear that the ACO approach can consistently find shorter makespans than any of the other approaches for all classes of ETC matrix tested, and this suggests that it would beat all the other approaches for the full problem suite. The ACO approach does, however, take significantly longer than any other approach, at around 3.5 hours it takes approximately 200 times longer than the GA.

These results show the effectiveness of the various techniques described above, and represent a spectrum of possible approaches to the problem. If a fast solution is required then Min-min combined with the local search mechanism can be used which will still outperform all previous solutions techniques tested here. If more time is available then the tabu search can be employed to improve on this solution, the more time is available the longer this can be left to run. Finally if the final makespan is critical and several hours (or even days) are available for scheduling the ACO algorithm can be used to find a very high quality solution.

These results are encouraging, and it is interesting to see how much better the approaches described here can perform than other current techniques. However I do

problem	Min-min	RC	GA	Min-min+LS	Min-min+Tabu	ACO
u-c-hihi.0	8460675.00	9576838.99	8050844.50	7711037.16	7568871.83	**7497200.85**
u-c-hilo.0	164022.44	163200.21	156249.20	154873.05	154644.48	**154234.63**
u-c-lohi.0	275837.34	309192.74	258756.77	251434.50	245981.55	**244097.28**
u-c-lolo.0	5546.26	5542.55	5272.25	5231.13	5202.51	**5178.44**
u-i-hihi.0	3513919.25	3447651.42	3104762.50	3021155.10	3021155.10	**2947754.12**
u-i-hilo.0	80755.68	76471.53	75816.13	74400.68	74400.68	**73776.24**
u-i-lohi.0	120517.71	126002.42	107500.72	104309.12	104309.12	**102445.82**
u-i-lolo.0	2779.09	2677.05	2614.39	2580.62	2580.62	**2553.54**
u-s-hihi.0	5160343.00	5068011.46	4566206.00	4256736.40	4248200.21	**4162547.92**
u-s-hilo.0	104540.73	101739.59	98519.40	97711.72	97711.72	**96762.00**
u-s-lohi.0	140284.48	143491.19	130616.53	126117.51	126115.39	**123922.03**
u-s-lolo.0	3867.49	3679.59	3583.44	3505.69	3505.69	**3455.22**

Table 6.3: Results for the first problem in each ETC matrix class, comparing the ACO approach with the other approaches. The ACO algorithm was allowed to run for 1000 iterations and took an average of 12792 seconds (just over 3.5 hours). The Min-min+Tabu search ran for an average of 12967 seconds. The best result is indicated in bold.

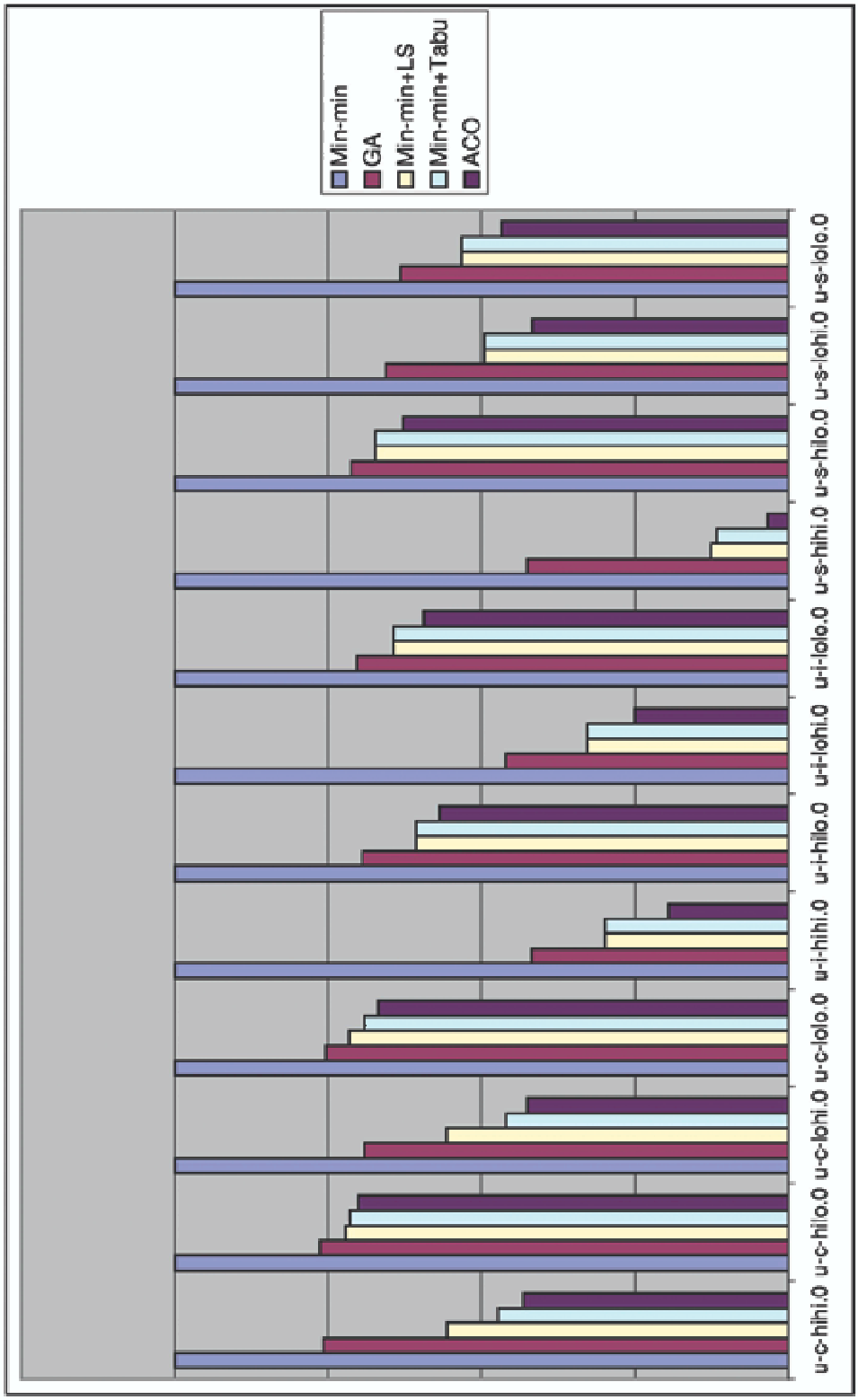

Figure 6.3: Normalised results (with the Min-min makespan set to 1) comparing the makespan found by the whole ACO algorithm with the other approaches. Note that the Y axis starts at 0.8.

not feel that the results achieved are very surprising. Of the three best techniques I have found in the literature, Min-min and the GA from [Braun et al., 2001] and RC from [Wu and Shu, 2001] two are fairly simple greedy heuristics, and experience has shown (from the BPP for example) that such heuristics can often be improved by local and tabu searches. The GA uses the Min-min solution to seed its initial population, and so it is to be expected that by a process of probabilistic search it should improve on this solution a little, and this is the result we see. Again, other research shows that a meta-heuristic approach such as a GA can often be improved with local search techniques such as those described here. The ACO approach which combines a probabilistic search guided by heuristic problem specific information and a simple form of information sharing about good solutions, with fast local search techniques would be expected to outperform such alternative techniques.

6.9.2 Comparing the components of the algorithm

Once again, as this is a hybrid approach it is interesting to compare the results of the algorithm used with and without certain components. A full run on each of the 1200 problem instances with several variations of the ACO algorithm would take a prohibitive amount of time, and as we only want to compare the algorithm with itself this is unnecessary. Results are therefore only provided for three, fairly representative, problem instances *u-c-hihi.0*, *u-i-hilo.0* and *u-s-lolo.0*. Table 6.4 shows the comparative results of running: Min-min with local and tabu searches applied to the solution, and the ACO algorithm without pheromone trail seeding, without local search, without tabu search, without local or tabu search, without using the pheromone trail (i.e. with an α value of 0) and finally the full algorithm. The tabu search was allowed to run for 1000 iterations, and each of the ACO runs were 100 iterations.

These results generally confirm the expected behaviour of each component, however there are a few interesting results to note. The run without pheromone trail seeding gets very close to the best result, and in one case beats the full algorithm. Pheromone trail seeding is mainly helpful to get the ants off to a good start, and the ant solutions for the first few iterations show that it does this well, but it seems that over the course of a full run it doesn't actually make a great difference. However it 'costs' nothing to

problem	Min-min	Min-min+LS	no LS or TS	no LS	no TS	no seeding	no pher	AntMPS
u-c-hihi.0	8460675.00	7711037.16	7711037.16	7584003.12	7695088.68	7559015.81	7662038.81	7558362.70
u-i-hilo.0	80755.68	80755.68	74400.68	74160.16	74400.68	73879.19	74400.68	74088.51
u-s-lolo.0	3806.83	3505.69	3504.74	3473.32	3501.23	3482.61	3499.92	3482.49

Table 6.4: Results comparing the behaviour of the various components of the algorithm on a few representative problem instances.

do and helps to find good solutions early on and so it is still used.

Another interesting result is that the run without local search finds results that are quite close to the whole algorithm, and finds a slightly better makespan for one problem. This shows that the tabu search (which is only run on the iteration best ant, not on every ant as for the local search) can effectively do the job of the local search, but the opposite is not true as the results without tabu search are consistently poorer than the full algorithm. However when both are used the best results are generally obtained.

The results without local or tabu search show the extent to which the ants rely on these, as the best solutions found are simply the initial solution generated by Min-min with local and tabu searches - no improvement is made by the ants.

The runs without using pheromone seem pretty competitive, but if the runs are analysed they show that the ants find very poor solutions, e.g. for *u-s-lolo* the ants find solutions above 4000, but the local and tabu searches can improve these to the results shown, which, again, demonstrates the power of these techniques.

6.9.3 Analysis of an example ACO run

To illustrate the behaviour of the ants over a complete run Appendix B shows some edited 'highlights' from a 1000 iteration run of the full program on the problem file *u-c-hihi.0*. Section 1 shows the run starting up, and displays the Min-min solution makespan and the improvements made to this by the local and tabu searches. Pheromone trail seeding is used in this run and so this solution will be immediately reinforced in the pheromone trail. Results from each iteration of the ACO algorithm are then displayed: the first value in each line is the makespan of the iteration best ant's solution, the next is the makespan after applying local search, and the final value in each line is the result of applying a tabu search of $nTrials$ iterations to this solution. An improvement to the global best solution (initially set to the tabu search solution above) is found immediately in iteration 0 (indicated by asterisks), helped by both the local and tabu searches. The ants quickly start finding improved solutions, and by iteration 8 an ant finds a sub-7600000 solution without any help from the local or tabu searches.

Section 2 shows an interesting behaviour that occurs in many runs: a new global

best solution is found and the ants then build identical solutions to it for a few iterations, before the tabu search finds a better solution (in iteration 38). This seems to 'encourage' the ants to start exploring again, probably by reinforcing some good, new job-processor pairings in the pheromone trail. These ant solutions may not be better then the global best, but the fact that the ants are exploring 'around' the solution is more useful than simply rebuilding the global best solution. Only the iteration best solution is shown, and so the other ants (9 in this case, because *numAnts* is set to 10) may be building alternative solutions while the iteration best is temporarily stuck.

Section 3 shows the ants usefully exploring around good solutions, and 3 new global best solutions are found in a few iterations. The local search helps to find 2 of these solutions, but for example in iteration 792, the tabu search does not improve the solution at all. This is quite common, but if the tabu search is not used much or at all (i.e. a low $nTrials$ value is used) the ants tend to get stuck building the same solution for many iterations, and the overall performance is significantly inhibited. In iterations 794-796 neither the local nor the tabu search improves the ant solution until an ant finds an alternative solution, which is quickly reduced to a local minimum (with a makespan equal to the solution previously found by the ants) by the local search.

Sections 4 and 5 show the final best solution being found and the run ending. It is interesting to note that the ants are continuing to explore and find different solutions even after nearly 1000 iterations. [Braun et al., 2001] note that their GA is usually stopped because there has been no change in the elite chromosome (equivalent to a global best solution here) in 150 iterations. This suggests that the GA has 'bottomed-out' and no matter how much longer it is run it will not find better solution. Here we can see that the ACO approach does not exhibit this undesirable behaviour and this behaviour can continue for even longer, in test runs of around 100,000 iterations the ants continue to find new solutions throughout the run. (The final solution found is not shown as with 512 processors it would fill several pages, the program does of course allow this to be displayed.)

Overall each of the components of the algorithm can be seen to be usefully performing its intended function. The ants build solutions, exploring the search space around promising areas and the local search can quickly take the ant's solutions to

their local optimum. The way in which that the ants continue to explore throughout the run is probably one of the greatest strengths of the approach. The tabu search cannot always improve on the local search but sometimes, using its ‘short-term memory’ it can overcome the local optimum. These improved solutions will then be reinforced, and the ants can use features of the new improved solutions to find solutions around them in the solution space, and so the process continues.

Chapter 7

Conclusions & Further Work

In this dissertation I have described two related approaches using ant colony optimisation and local search techniques to two important versions of the multi-processor scheduling problem. In this chapter I briefly discuss possible further work improving the algorithms described, and discuss possible ways of applying such techniques to other related scheduling problems, and finally provide my overall conclusions about results that were obtained.

7.1 Further work

The two versions of the MPSP dealt with in this dissertation are part of a much larger set of related scheduling problems, and there are several other important formulations of the MPSP alone. It is hoped that the concepts presented in the dissertation could be successfully applied to some other formulations and two in particular are addressed in this section. This section also suggests some possible areas in which the approaches presented in this dissertation could be improved.

7.1.1 Improvements to the current approaches

As mentioned briefly earlier I think that a lot more experimentation could be performed simply with the parameters of the ACO algorithms described here. The parameter values used are admittedly ‘heuristic’ guesses at good values, and while they do give

reasonable results, a more comprehensive test suite and analysis could significantly improve performance.

The local searches used with both ACO algorithms are fairly simple, and improved strategies may help significantly. This may be particularly true for the homogeneous problem, as the ants can generally find very good quality solutions, but often cannot quite reach the optimum. A local search that could improve even a little on the one used here may significantly improve overall results.

In a similar vein, the tabu search technique employed for the heterogeneous problem is a fairly simple implementation of the concept. Much research has been performed on ways to improve the performance of TS algorithms, and experimentation with some of these could improve the, already promising, results in the heterogeneous scheduling domain. An effective tabu search could perhaps be also be used to help the ants along in the homogeneous problem.

Further experimentation with the ant solution building process used for the heterogeneous problem might also be useful. The current scheme is admittedly rather inflexible, but as the results demonstrate, it does seem to be effective.

7.1.2 Dealing with job precedence

Another important variation of the MPSP is the problem of scheduling jobs which have inter-job precedence constraints. Many real-world scheduling problems require that some job j_a must have finished before some other job j_b can run, for example. As a result of these inter-job relations such problems are typically represented as directed acyclic graphs (DAGs), in which the nodes represent tasks and the arcs represent communication between the jobs. Jobs have an associated running time, and the arcs may have a communication cost associated with them, an example DAG is shown in figure 7.1.

This is clearly a rather different problem than the two problems studied here and has been approached with various techniques (see e.g. [Kwok and Ahmad, 1999] for a comparison of many of the important techniques).

I have implemented a ‘proof of concept’ ACO algorithm for scheduling jobs with precedence constraints. The heuristic foundation for the ant solution building pro-

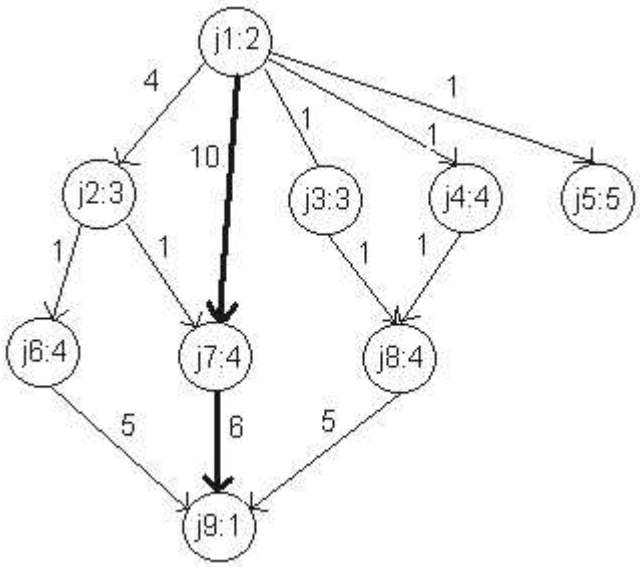

Figure 7.1: An example DAG, a node *jn*:*t* represents a job *n* with a running time of *t*, and the arcs represent communication requirement between the jobs, and have an associated cost(figure after [Kwok and Ahmad, 1999]).

node	**s. level**
1	11
2	8
3	8
4	9
5	5
6	5
7	5
8	5
9	1

Table 7.1: Showing the *static level* of each node in figure 7.1

cess is one of many 'list-scheduling' algorithms which address the problem by simply sorting all the jobs that are currently runnable - those whose predecessors have all completed - by their *priority*, and then allocating the job with the highest priority to the first available, i.e. idle, processor. Several different techniques for determining a job's priority are used (see e.g. [Kwok and Ahmad, 1999] for a review). A simple one to compute which gives reasonable results, is according to the *static level* (sl) of a job. This is defined as the longest path, in terms of execution time, from that job to an exit node. The static levels for each job in figure 7.1 are shown in table 7.1. The heuristic simply schedules the runnable job with the highest static level first. This heuristic, therefore, aims to minimise the makespan by scheduling jobs that have a long running time, or have a long-running dependent jobs, or combination of these, early on. This measure does not take the communication costs into account, but other priority measurement schemes do.

In the ACO algorithm the heuristic value of each job is simply its static level, i.e. $\eta(j) = sl(j)$. Firstly the set R of ready jobs is assembled, and then a job j from this set is picked probabilistically with respect to this heuristic value, as shown in equation 7.1.

$$prob(j) = \frac{[\eta(j)]^{\beta}}{\sum_{i=1}^{|R|}[\eta(j_i)]^{\beta}} \forall j_i \in R \tag{7.1}$$

Once the job to be scheduled next, j_c, has been selected, the set of idle processors I is established, a processor p from this set to which j_c will be scheduled is selected probabilistically with respect to the pheromone between j_c and it, as shown in equation 7.2

$$prob(p) = \frac{[\tau(p, j_c)]^{\alpha}}{\sum_{i=1}^{n}[\tau(p_i, j_c)]^{\alpha}} \forall p_i \in I \tag{7.2}$$

If no idle processor is available the algorithm simply waits until one becomes available. This process if then repeated until all the jobs have been scheduled. The pheromone update is handled in exactly the same way as for the two problems described earlier.

This is a fairly simple ACO implementation, but brief experimentation (using the benchmark Standard Task Graph set[1]) shows that it can find shorter makespans than a greedy static level heuristic on some problem instances, and can occasionally find optimal solutions.

Although the ant solution building process and the pheromone trail are defined in similar ways to the two ACO algorithms described earlier, the underlying features of the problem are rather different and so the local and tabu search techniques used before could not be directly applied to this problem, and there was not enough time available to investigate and implement a new local search strategy. The brief results so far do seem to indicate that an ACO approach to this problem may be very successful if a suitable fast local search could be developed. There is also a lot of scope for experimenting with alternative list scheduling strategies, and perhaps also using entirely different heuristic approaches.

7.1.3 Dynamic scheduling

Both the MPSP problems dealt with here, and the job precedence problem discussed above, are static problems in which all the jobs to be scheduled are known *a priori*, but often this information is not available and jobs must be *dynamically* scheduled as they appear on the system. [Maheswaran et al., 1999] provide a comparison of some approaches to dynamically scheduling independent jobs onto a heterogeneous computing system using ETC matrices as were used in this dissertation. They divide the approaches up into two main types: *on-line*, which schedule jobs immediately as they arrive at the scheduler, and *batch*, which waits for some period of time and then schedules a 'batch' of ready jobs at once. The batch mode heuristics are essentially static scheduling heuristics which are used after a timeout (set to 10 seconds in the study) to schedule all newly arrived jobs. It therefore seems that the static techniques described in this dissertation could be readily applied to a batch mode dynamic scheduling system, although the ACO approach and the longer TS runs would probably take too long to be used in practice.

[Maheswaran et al., 1999] again find that the Min-min heuristic is fairly fast

[1] available at: http://www.kasahara.elec.waseda.ac.jp/schedule/index.html

and effective for this problem, but it is beaten by a newer heuristic they propose called the 'sufferage' heuristic (SH). I have re-implemented SH (for details see [Maheswaran et al., 1999]) and tested on the ETC matrices used in chapter 6. My implementation does not appear to perform better on these entire instances than Min-min, but it is not designed to be used for such large ETC matrices, so by simply reducing the number of jobs to be scheduled we give it a (slightly) more realistic problem. Tests using only 64 jobs (and 16 processors as before) show that SH does often outperform Min-min on smaller problems. To illustrate the way the local and tabu searches could be applied to solutions produced by SH brief test results for scheduling the first 64 jobs from the first problem in each ETC matrix class are shown in table 7.2. For these results the tabu list size was set to 10 and the tabu search ran for 1000 iterations.

It should be noted that these results are *not* directly comparable to the results provided in [Maheswaran et al., 1999] as they use a complex event simulator to simulate a real dynamic scheduling environment which includes such features as varying the actual running time of each job compared to the ETC time, and modelling jobs arriving in bursts etc. However these results do suggest that the local and tabu searches may successfully improve the schedules found by the Min-min and SH heuristics used in a batch-mode dynamic scheduler. The local search is certainly quick enough to be usefully employed, and with a scheduler time delay of 10 seconds as is used in [Maheswaran et al., 1999], the tabu search is probably also fast enough.

This is a fairly simple, albeit useful, addition to current methods but it doesn't really use any of the ACO concepts. ACO algorithms have, however, been successfully applied to many dynamic problems, a good example is the AntNet system described in [Bonabeau et al., 1999] which is used for routing in a packet-switched telecommunications networks. This works essentially by using a process of pheromone reinforcement to identify good routes from one point to another in the network, which can change as load on the network varies (reminiscent of the original biological ant experiment). It may be that a distributed ACO algorithm like AntNet could be employed in a heterogeneous (or even homogeneous) network to efficiently distribute jobs across the machines in the network, avoiding busy regions. However I think it would require a rather different approach to the ACO algorithms used here.

	Min-min						Sufferage					
	Init		LS		TS		Init		LS		TS	
problem	ms	time	ms	time	ms	time	ms	time	ms	time	ms	time
u-c-hihi.0	1760833.29	0.00	1281095.56	0.06	**1249888.73**	1.48	1646667.56	0.05	1461943.25	0.00	1251242.13	1.21
u-c-hilo.0	28912.38	0.00	24429.20	0.00	22419.37	1.43	27062.69	0.05	23091.48	0.00	**22164.96**	1.21
u-c-lohi.0	57232.30	0.06	42061.37	0.00	41619.25	1.54	62706.13	0.05	52613.48	0.00	**40306.18**	1.27
u-c-lolo.0	998.92	0.05	771.26	0.00	**740.09**	1.43	880.11	0.00	770.37	0.06	740.41	1.42
u-i-hihi.0	703868.21	0.05	565575.25	0.00	565575.25	1.32	619191.40	0.05	565575.25	0.00	**549991.05**	1.21
u-i-hilo.0	12176.13	0.00	10298.82	0.00	9935.96	1.37	10527.72	0.00	**9610.98**	0.00	**9610.98**	1.37
u-i-lohi.0	23604.18	0.05	19683.95	0.00	**19683.95**	1.37	21160.64	0.00	20844.77	0.00	**19683.95**	1.37
u-i-lolo.0	493.57	0.06	445.01	0.00	**418.61**	1.37	541.95	0.00	**418.61**	0.00	**418.61**	1.37
u-s-hihi.0	1190978.03	0.05	872214.06	0.00	**766326.31**	1.43	1090510.22	0.00	848478.41	0.00	**766326.31**	1.27
u-s-hilo.0	21627.07	0.00	15529.14	0.00	15409.63	1.43	15935.55	0.00	**15027.31**	0.00	**15027.31**	1.43
u-s-lohi.0	30529.32	0.06	19808.53	0.00	**19808.53**	1.43	28839.41	0.00	24003.64	0.00	20341.93	1.43
u-s-lolo.0	925.05	0.06	616.20	0.00	604.61	1.37	792.78	0.05	**599.37**	0.00	**599.37**	1.38

Table 7.2: Results of running the local and tabu searches on the solutions found by Min-min and the Sufferage heuristic scheduling the first 64 jobs in each ETC matrix onto 16 processors. The best result for each problem is shown in bold.

7.2 Conclusions

The results for the reformulated BPP problems show that the ACO meta-heuristic can be effectively adapted to deal with the homogeneous MPSP. The ACO algorithm with local search can find optimal solutions for many different problems, but it does struggle to find the optimum for some classes of problems, and cannot outperform refined BPP solution techniques. It is however fairly robust, as even when an optimal solution is not found, generally good quality solutions are. It is unfortunate that no MPSP benchmark problems were available for this problem as it would be interesting to compare against other, purpose-designed techniques.

The results for the heterogeneous problem are rather more encouraging. The local and tabu search techniques described are very effective, and alone they can outperform all the alternative techniques that I have found in the literature. When combined with an ACO algorithm even better solutions can be found, although it does take significantly longer. This was to be expected though, ACO algorithms work by building multiple solutions over several generations, and so are unlikely to be competitive in terms of time with 'single sweep' solution building techniques such as Min-min. The range of new techniques described appear to be complimentary, but I think they also represent a useful 'toolkit' for this problem which future applications could use depending on the constraints encountered in a particular domain.

In conclusion I feel that I have demonstrated that the ACO meta-heuristic, when combined with local search techniques, can be successfully applied to the multi-processor scheduling problem. Two hybrid ACO algorithms for static homogeneous and heterogeneous multi-processor scheduling have been designed and implemented, and experiments show they are successful, particularly for the heterogeneous problem. I have also suggested two possible approaches to applying similar techniques to two further variations of the MPSP. I think that this suggests that there is considerable scope for future research in this area, and also demonstrates the potential of applying the ACO meta-heuristic on problems that have moved some distance from those encountered by real ants!

Appendix A

Example output from an AntMPS for homogeneous processors run

This is the complete output from a single run of AntMPS on the reformulated bin-packing problem *u120-08* from [Falkenauer, 1996].

```
File: c:/prob/bp/u120_08mps.txt, alpha: 20.0, beta: 20.0, gamma: 0.0
delta: 2.0, nAnts: 10, nProcessors: 50, trailMin: 0.01, nJobs: 120
maxIterations: 1000

Ideal makespan: 150, FFD makespan: 160.0 (FFD time: 0.0s)

FFD makespan after local search: 151.0 (search took: 0.05s)

Iteration 0: 161.0 ++ 0.8993237085711545 --> 151.0 ++ 0.9886633271816504
Iteration 1: 161.0 ++ 0.9232230155101466 --> 151.0 ++ 0.9893348301992703
Iteration 2: 155.0 ++ 0.9572509382416997 --> 151.0 ++ 0.9893778104065338
Iteration 3: 154.0 ++ 0.9680351971985903 --> 151.0 ++ 0.9895764612745744
Iteration 4: 153.0 ++ 0.9752201211130512 --> 151.0 ++ 0.9897070467141726
Iteration 6: 152.0 ++ 0.9829041541407678 --> 151.0 ++ 0.9898376666226739
Iteration 8: 155.0 ++ 0.960854650797967 --> 151.0 ++ 0.9899683210137274
```

```
Iteration 11: 154.0 ++ 0.9670136513382942 --> 151.0 ++ 0.9896857242883689
Iteration 17: 152.0 ++ 0.9812579727210283 --> 150.0 ++ 1.0

Best ant found: [
        P1 t=150: [(J83:79), (J29:42), (J12:29)],
        P2 t=149: [(J116:98), (J41:51)],
        P3 t=149: [(J100:89), (J58:60)],
        P4 t=149: [(J56:58), (J101:91)],
        P5 t=150: [(J43:52), (J63:64), (J19:34)],
        P6 t=149: [(J119:100), (J40:49)],
        P7 t=149: [(J20:35), (J107:94), (J2:20)],
        P8 t=149: [(J112:97), (J45:52)],
        P9 t=150: [(J94:86), (J24:38), (J8:26)],
        P10 t=150: [(J96:88), (J61:62)],
        P11 t=150: [(J87:81), (J72:69)],
        P12 t=150: [(J16:33), (J111:96), (J3:21)],
        P13 t=150: [(J5:23), (J49:57), (J73:70)],
        P14 t=150: [(J117:98), (J44:52)],
        P15 t=149: [(J99:89), (J59:60)],
        P16 t=150: [(J97:88), (J26:39), (J6:23)],
        P17 t=150: [(J22:37), (J27:40), (J77:73)],
        P18 t=150: [(J4:22), (J108:94), (J18:34)],
        P19 t=149: [(J89:82), (J69:67)],
        P20 t=149: [(J53:58), (J104:91)],
        P21 t=150: [(J17:33), (J1:20), (J114:97)],
        P22 t=149: [(J92:85), (J64:64)],
        P23 t=149: [(J110:95), (J47:54)],
        P24 t=149: [(J82:77), (J76:72)],
        P25 t=150: [(J86:81), (J70:69)],
        P26 t=150: [(J81:76), (J32:45), (J13:29)],
        P27 t=150: [(J78:74), (J35:46), (J14:30)],
        P28 t=150: [(J93:85), (J66:65)],
        P29 t=149: [(J109:95), (J48:54)],
```

```
        P30 t=150: [(J31:44), (J37:48), (J54:58)],
        P31 t=149: [(J118:100), (J39:49)],
        P32 t=150: [(J25:39), (J90:84), (J10:27)],
        P33 t=149: [(J115:98), (J42:51)],
        P34 t=150: [(J57:59), (J103:91)],
        P35 t=149: [(J102:91), (J55:58)],
        P36 t=150: [(J91:85), (J65:65)],
        P37 t=150: [(J113:97), (J46:53)],
        P38 t=149: [(J50:57), (J105:92)],
        P39 t=150: [(J98:89), (J60:61)],
        P40 t=150: [(J15:32), (J30:43), (J80:75)],
        P41 t=150: [(J62:63), (J36:47), (J28:40)],
        P42 t=149: [(J120:100), (J38:49)],
        P43 t=150: [(J67:67), (J52:57), (J9:26)],
        P44 t=150: [(J85:80), (J74:70)],
        P45 t=150: [(J71:69), (J34:45), (J21:36)],
        P46 t=149: [(J51:57), (J106:92)],
        P47 t=149: [(J23:37), (J95:87), (J7:25)],
        P48 t=149: [(J33:45), (J11:29), (J79:75)],
        P49 t=150: [(J84:79), (J75:71)],
        P50 t=149: [(J88:82), (J68:67)]]

File: c:/prob/bp/u120_08mps.txt

Optimal solution found

makespan: 150.0, iteration: 17, time: 4.94s, average local search depth: 5
```

Appendix B

Example output from an AntMPS for heterogeneous processors run

This is edited output from a run of AntMPS on the problem instance *u-c-hihi.0* from [Braun et al., 2001].

```
<section 1>
_____________________________________________________________________

File: /home/s0237680/prob/hp/u_c_hihi.0, alpha: 10.0, beta: 10.0, gamma: 0.0
nAnts: 10, trailMin: 0.01, nJobs: 512, nProcessors: 16, nTrials: 1000
maxIterations: 1000, useJobTimes: false, seedTrail: true

Min-min makespan: 8460675.003243 (Min-min time: 0.303)

Min-min makespan after local search: 7704970.3654849995 (search took: 0.192)

Min-min makespan after tabu search: 7637683.971627 (search took: 8.773)

*** Iteration 0: 7676165.016416994 --> 7634360.596036995 --> 7626414.489936994
Iteration 1: 7967787.361021995 --> 7665970.870482002 --> 7656245.282164996
*** Iteration 2: 7652865.362914001 --> 7625392.463869 --> 7625392.463869
*** Iteration 3: 7629171.046876 --> 7616922.914854995 --> 7613359.045956995
```

```
*** Iteration 4: 7691932.4517499935 --> 7605512.613037999 --> 7605512.613037999
*** Iteration 5: 7628013.3833899945 --> 7604598.262205998 --> 7604598.262205998
*** Iteration 6: 7625399.419555999 --> 7598298.384398999 --> 7598298.384398999
Iteration 7: 7628513.9204689935 --> 7600717.600097999 --> 7600717.600097999
*** Iteration 8: 7598298.384398999 --> 7595895.293545993 --> 7595895.293545993
Iteration 9: 7596892.762295993 --> 7596875.064940998 --> 7596875.064940998
Iteration 10: 7600714.772093998 --> 7596484.980312994 --> 7596484.980312994

<section 2>

*** Iteration 30: 7584284.728085994 --> 7584284.728085994 --> 7584284.728085994
Iteration 31: 7584284.728085995 --> 7584284.728085995 --> 7584284.728085995
Iteration 32: 7584284.728085995 --> 7584284.728085995 --> 7584284.728085995
Iteration 33: 7584284.728085995 --> 7584284.728085995 --> 7584284.728085995
Iteration 34: 7584284.728085995 --> 7584284.728085995 --> 7584284.728085995
Iteration 35: 7584284.728085994 --> 7584284.728085994 --> 7584284.728085994
Iteration 36: 7584284.728085995 --> 7584284.728085995 --> 7584284.728085995
Iteration 37: 7584284.728085994 --> 7584284.728085994 --> 7584284.728085994
*** Iteration 38: 7584284.728085997 --> 7584284.728085997 --> 7581667.334774997
Iteration 39: 7701670.975217996 --> 7636315.5967999995 --> 7636315.5967999995
Iteration 40: 7605980.763363996 --> 7585758.191407 --> 7585758.191407
Iteration 41: 7685502.926756997 --> 7623614.291442997 --> 7623614.291442997

<section 3>

Iteration 789: 7506617.028806999 --> 7506617.028806999 --> 7506617.028806999
*** Iteration 790: 7509247.944973994 --> 7506606.18701 --> 7506606.18701
Iteration 791: 7506617.028806999 --> 7506617.028806999 --> 7506617.028806999
*** Iteration 792: 7507188.955075999 --> 7506606.1870099995 --> 7506606.1870099995
Iteration 793: 7506606.1870099995 --> 7506606.1870099995 --> 7506606.1870099995
*** Iteration 794: 7506606.187009999 --> 7506606.187009999 --> 7506606.187009999
Iteration 795: 7506606.1870099995 --> 7506606.1870099995 --> 7506606.1870099995
Iteration 796: 7506606.1870099995 --> 7506606.1870099995 --> 7506606.1870099995
```

```
Iteration 797: 7512222.705136994 --> 7506606.1870099995 --> 7506606.1870099995

<section 4>

Iteration 973: 7497759.475551995 --> 7497604.121094 --> 7497604.121094
Iteration 974: 7539978.090633996 --> 7508965.470302994 --> 7508965.470302994
*** Iteration 975: 7497200.850185994 --> 7497200.850185994 --> 7497200.850185994
Iteration 976: 7518141.137783995 --> 7507100.39807 --> 7507100.39807
*** Iteration 977: 7497200.850185993 --> 7497200.850185993 --> 7497200.850185993
Iteration 978: 7497200.850185995 --> 7497200.850185995 --> 7497200.850185995
Iteration 979: 7497200.850185996 --> 7497200.850185996 --> 7497200.850185996

<section 5>

Iteration 996: 7542285.019222993 --> 7517805.973019994 --> 7514537.077053995
Iteration 997: 7497200.850185994 --> 7497200.850185994 --> 7497200.850185994
Iteration 998: 7497200.850185994 --> 7497200.850185994 --> 7497200.850185994
Iteration 999: 7897881.934382996 --> 7538161.081693994 --> 7538161.081693994

Best makespan found: 7497200.850185993, Iteration: 977, Time: 18600.874s
```

Bibliography

[Abraham et al., 2000] Abraham, A., Buyya, R., and Nath, B. (2000). Nature's heuristics for scheduling jobs on computational grids. available from: http://citeseer.nj.nec.com/article/abraham00natures.html.

[Alvim et al., 2002] Alvim, A. C. F., Glover, F., Ribiero, C. C., and Aloise, D. J. (2002). A hybrid improvement heuristic for the bin packing problem. available from: http://citeseer.nj.nec.com/557429.html.

[Bonabeau et al., 1999] Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York.

[Braun et al., 2001] Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837.

[Corcoran and Wainwright, 1994] Corcoran, A. L. and Wainwright, R. L. (1994). A parallel island model genetic algorithm for the multiprocessor scheduling problem. In *Selected Areas in Cryptography*, pages 483–487.

[Dorigo, 1992] Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms*. PhD thesis, DEI, Polytecnico di Milano, Milan, Italy. (in Italian).

[Dorigo and Stützle, 2002] Dorigo, M. and Stützle, T. (2002). The ant colony optimization metaheuristic: Algorithms, applications, and advances. In Glover, F. and

Kochenberger, G., editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research and Management Science*, pages 251–285. Kluwer Academic Publishers.

[Ducatelle, 2001] Ducatelle, F. (2001). Ant colony optimisation for bin packing and cutting stock problems. Master's thesis, University of Edinburgh.

[Falkenauer, 1996] Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30.

[Fox and Long, 2001] Fox, M. and Long, D. (2001). Multi-processor scheduling problems in planning. International Conference on AI, IC-AI'01, Las Vegas.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. (1979). *Computers and Intractability: A Guide to the theory of NP-Completeness*. Freeman and Company, San Francisco.

[Gent, 1998] Gent, I. P. (1998). Heuristic solution of open bin packing problems. *Journal of Heuristics*.

[Glover and Hübscher, 1994] Glover, F. and Hübscher, R. (1994). Applying tabu search with influential diversification to multiprocessor sceduling. *Computers in Operations Research*, 21:877–844.

[Glover and Laguna, 1997] Glover, F. and Laguna, M. (1997). *Tabu Search*. Kluwer Academic publishers, Boston.

[Kwok and Ahmad, 1999] Kwok, Y.-K. and Ahmad, I. (1999). Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422.

[Levine and Ducatelle, 2003] Levine, J. and Ducatelle, F. (2003). Ant colony optimisation and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*. (forthcoming).

[Maheswaran et al., 1999] Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., and Freund, R. F. (1999). Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59:107–131.

[Martello and Toth, 1990] Martello, S. and Toth, P. (1990). *Knapsack Problems, Algorithms and Computer Implementations*. John Wiley and Sons Ltd., England.

[Mitchell, 1998] Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

[Nath et al., 1998] Nath, B., Lim, S., and Bignall, R. J. (1998). A genetic algorithm for scheduling independent jobs on uniform machines with multiple objectives. In *Proceedings of the International Conference on Computational Intelligence and Multimedia applications*, pages 67–74, Australia.

[Reeves, 1996] Reeves, C. (1996). Hybrid genetic algorithms for bin-packing and related problems. *Annals of the Operation Research Society*, 63:371–396.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey.

[Schoenfield, 2003] Schoenfield, J. E. (2003). personal communication.

[Selman et al., 1993] Selman, B., Kautz, H., and Cohen, B. (1993). Local search strategies for satisfiability testing. In Johnson, D. S. and Trick, M. A., editors, *Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability*, Providence RI.

[Stützle and Hoos, 2000] Stützle, T. and Hoos, H. (2000). Max-min ant system. *Future Generation Computer Systems*, 16(8):889–914.

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.

[Thiesen, 1998] Thiesen, A. (1998). Design and evaluation of tabu search algorithms for multiprocessor scheduling. *Journal of Heuristics*, 4:141–160.

[van der Zwaan and Marques, 1999] van der Zwaan, S. and Marques, C. (1999). Ant colony optimisation for job shop scheduling. In *Proceedings of the Third Workshop on Genetic Algorithms and Artificial Life (GAAL 99)*.

[Vink, 1997] Vink, M. (1997). Solving combinatorial problems using evolutionary algorithms. available from: http://citeseer.nj.nec.com/vink97solving.html.

[Wu and Shu, 2001] Wu, M. and Shu, W. (2001). A high-performance mapping algorithm for heterogeneous computing systems. International Parallel and Distributed Processing Symposium (IPDPS).