A fast, effective local search for scheduling independent jobs in heterogeneous computing environments

Graham Ritchie and **John Levine**

Centre for Intelligent Systems and their Applications
School of Informatics, University of Edinburgh
Appleton Tower, Crichton Street, Edinburgh, EH8 9LE
Graham.Ritchie@ed.ac.uk

Abstract

The efficient scheduling of independent computational jobs in a heterogeneous computing (HC) environment is an important problem in domains such as grid computing. Finding optimal schedules for such an environment is (in general) an NP-hard problem, and so heuristic approaches must be used. Work with other NP-hard problems has shown that solutions found by heuristic algorithms can often be improved by applying local search procedures to the solution found. This paper describes a simple but effective local search procedure for scheduling independent jobs in HC environments which, when combined with fast construction heuristics, can find shorter schedules on benchmark problems than other solution techniques found in the literature, and in significantly less time.

Introduction & Motivation

The efficient scheduling of independent computational jobs in a heterogeneous computing (HC) environment such as a computational grid is clearly important if good use is to be made of such a valuable resource. However finding optimal schedules in such a system has been shown, in general, to be NP-hard (it is a generalised reformulation of SS8 from (Garey & Johnson 1979)).

Static scheduling algorithms can be used in such a system for several different requirements (Braun *et al.* 2001). The first, and most common, 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 subnetwork 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, static scheduling techniques can also be applied to a dynamic scheduling problem. If the static scheduler is fast enough that it could be run in 'batch mode' every few seconds to schedule jobs that have arrived on the system since the last scheduler run. This may allow a dynamic scheduler to make better decisions (Maheswaran *et al.* 1999), as instead of simply scheduling each task as it arrives, a batch mode dynamic scheduler has more resource requirement information at once and so can make more informed decisions.

	processor 1	processor 2
job 1	2	3
job 2	3	4
job 3	4	5
job 4	5	6

Table 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, as processor 1 is consistently faster than processor 2.

Simulation Model

Real-world HC systems, such as a computational grid, are complex combinations of 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) describes a benchmark simulation model for comparison of static scheduling algorithms for HC environments. They define the notion of a *metatask* as a collection of independent jobs with no inter-job 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 is assumed to be available *a priori*. Essentially, the expected running time of each individual 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 4 jobs and 2 processors is given in table 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 in the experiments described below.

Current techniques

Static scheduling

(Braun *et al.* 2001) provides a comparison of 11 static heuristics for scheduling in HC environments, and the reader is referred there for details of the various schemes that are used. A range of simple greedy construction heuristic approaches are compared and some of these are briefly described below.

OLB *Opportunistic Load Balancing* assigns each job in arbitrary order to the processor with the shortest schedule, irrespective of the ETC on that processor. OLB is intended to try to balance the processors, but because it does not take execution times into account it finds rather poor solutions.

MET *Minimum Execution Time* assigns each job in arbitrary order to the processor on which it is expected to be executed fastest, regardless of the current load on that processor. MET tries to find good job-processor pairings, but because it does not consider the current load on a processor it will often cause load imbalance between the processors.

MCT *Minimum Completion Time* assigns each job in arbitrary order to the processor with the minimum expected *completion time* for the job. The completion time of a job j on a processor p is simply the ETC of j on p added to p's current schedule length. This is a much more successful heuristic as both execution times and processor loads are considered.

Min-min establishes the minimum completion time for every unscheduled job (in the same way as MCT), and then assigns the job with the *minimum* minimum completion time (hence Min-min) to the processor which offers it this time. Min-min uses the same intuition as MCT, but because it considers the minimum completion time for all jobs at each iteration it can schedule the job that will increase the overall makespan the least, which helps to balance the processors better than MCT.

Max-min is very similar to Min-min. Again the minimum completion time for each job is established, but the job with the *maximum* minimum completion time is assigned to the corresponding processor. Max-min is based on the intuition that it is good to schedule larger jobs earlier on so they won't 'stick out' at the end causing a load imbalance. However experimentation shows that Max-min cannot beat Min-min on any of the test problems used here.

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 that 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 have been 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).

Dynamic scheduling

(Maheswaran *et al.* 1999) describes several heuristics for dynamically scheduling independent jobs in HC environments. The approaches are separated into two main groups: *immediate mode* heuristics, which schedule jobs immediately as they arrive at the scheduler, and *batch mode* heuristics, which wait for some period of time and then schedule a 'batch' of ready jobs at once. The simulation model used in this study uses the same ETC matrix types as (Braun *et al.* 2001), but also includes a discrete event simulator which models the job arrival rate and other features, such as modifying the actual execution time of a given job.

Local search cannot be combined with the immediate mode heuristics, and so these are not discussed here. The batch mode heuristics, however, are essentially static scheduling heuristics which are used after a timeout (either a fixed time interval or after some fixed number of jobs have arrived) to schedule all newly arrived jobs. As noted earlier using a batch mode heuristic may allow the scheduler to make more informed decisions. The Min-min heuristic is again tested and performs well, but is slightly outperformed by a newly proposed algorithm known as the *Sufferage* heuristic (SH). SH essentially allocates a job to a processor if that job would 'suffer' most were it not allocated to that processor. The *sufferage value* of a job j_i is defined as the difference between its earliest completion time (on some processor p_a) and its second earliest completion time (on some other processor p_b), i.e. allocating j_i to p_a will give the best completion time for j_i, and allocating it to p_b will give the second best completion time. The algorithm is briefly described below (for a more detailed description see (Maheswaran *et al.* 1999)).

Firstly all processors are marked as unassigned. An arbitrary job j_k from the list of unallocated jobs is selected and the processor p_m that gives the earliest completion time for j_k is established. If p_m is unassigned then j_k is tentatively allocated to p_m, p_m is marked as assigned, and j_k is removed from the list of unallocated jobs. If, however, p_m has already been assigned to some other job j_l then j_l is unallocated from p_m and added back into the list of unallocated jobs. The sufferage value of j_k and j_l is then established and the job that has the highest value is selected, allocated to p_m and removed from the list of unallocated jobs. The job that was not selected will not be considered again until the next iteration. Once all jobs have been considered this process is repeated until all jobs have been allocated. The results provided in (Maheswaran *et al.* 1999) shows that SH generally outperforms both Min-min and immediate mode techniques for their test problems.

The local search procedure

Work in related NP-hard problems, such as bin packing (e.g. (Alvim *et al.* 1999), (Levine & Ducatelle 2003)), suggests that solutions built by many heuristic techniques can often be improved by using a local search procedure to take the solution to its local optimum in the search space. The important consideration in any local search procedure is how to define the *neighbourhood* of a solution. there are many possible neighbourhood definitions for this problem, but an obvious one is simply to consider solutions which differ by some number of job assignments. The procedure used here is described below.

In any solution there will be one or more 'problem' processors - those with schedule lengths equal to the makespan of the whole solution - and it seems sensible to try to reduce their makespan as this will immediately reduce the overall makespan of the solution. The local search procedure therefore considers all solutions which differ by swapping a job from a problem processor with another job from another processor. This means that for any solution there will be $a \times (n - a)$ neighbouring solutions according to this definition (where a is the number of jobs assigned to the problem processor).

Considering job swaps between processors is effective, but as only single job pair swaps are considered the number of jobs assigned to any processor will remain constant. For some solutions it may also be useful to consider job transfers from one processor to another as well (as noted in (Thiesen 1998)). The local search algorithm therefore also considers solutions which differ by transferring a single job from a problem processor to another processor. For this case there will be $a \times (m - 1)$ neighbouring solutions.

The neighbourhood of a solution is therefore defined as all solutions which differ from the original solution by a single job transfer from a problem processor to another processor, or which differ by a single job pair swap. This leaves us with a total neighbourhood size of $(a \times (n-a)) + (a \times (m-1))$. Some of these neighbouring solutions will have a longer makespan than the original solution and so these are ignored. This leaves us with a subset of the neighbouring solutions which have a smaller makespan than the original solution. Some local search procedures simply select the first improved solution that is found - a 'first-ascent approach'. Experimentation showed that exhaustively considering all improved neighbours and then selecting the one that reduces the maximum makespan of the two processors the most (a 'best-ascent' approach) worked best here. The procedure as a whole therefore exhaustively analyses the entire neighbourhood of the solution and selects the best neighbour as the new solution. This process is then repeated until no improved neighbour can be found. A simple example of this local search is shown in figure 1.

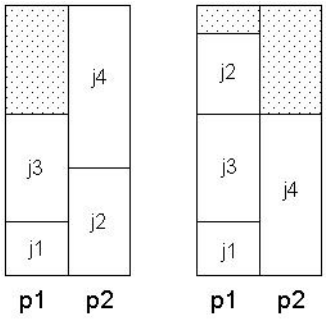

Figure 1: A simple example of the local search. The solution on the left hand side shows the Min-min solution to the ETC matrix shown in table 1, this has a makespan of 10. The local search finds that the solution formed by transferring *j2* from *p2* to *p1* (as shown in the solution on the right hand side) which reduces the makespan to 9, is the best neighbour and so this is the solution selected. There are alternative neighbours with makespans of 9 but the procedure does not distinguish between these and so the first such solution found in selected. No improved neighbouring solution can then be found, and so the local search terminates. Note, however, that this is only a local optimum, and is not the globally optimal solution, as *j2* and *j4* allocated to *p1*, and *j1* and *j3* allocated to *p2* would have a makespan of 8.

Experimental results

Static scheduling

We applied the local search to solutions produced by a number of heuristics and to solutions picked at random. Experimentation showed that using Min-min to generate an initial solution produced the best results overall. Results are therefore shown for running Min-min and then the local search procedure described above, compared with the GA and Min-min results from (Braun *et al.* 2001). Tests were performed on 1.6-2 GHz Linux machines, and all programs were written in Java.

Table 2 shows the average makespans found and average time taken (in seconds) by each technique on the 100 instances of each ETC matrix class. The different classes and 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.

It is clear from these results that the local search can effectively improve the Min-min solution, finding significantly shorter makespans than the GA, the best technique found in (Braun *et al.* 2001). The time taken by using Min-min and then the local search is consistently under half a second, and so even taking into account differences in machine speeds, this combination is significantly faster than the GA. The local search does not seem to be affected by the class of ETC matrix used - it consistently finds shorter makespan for all classes tested.

It should be noted that although combining Min-min and the local search gave the best results overall, the local search procedure can effectively improve solutions generated by every method tested, although it does take significantly

	Min-min		GA		Min-min+LS	
type	makespan	time	makespan	time	makespan	time
u-c-hihi	8428258.43	0.17	7906149.42	63.88	**7667606.81**	0.49
u-c-hilo	162745.18	0.17	155604.90	64.65	**153990.94**	0.39
u-c-lohi	283083.40	0.16	266723.49	62.99	**258656.25**	0.49
u-c-lolo	5460.25	0.15	5221.13	65.01	**5167.83**	0.38
u-i-hihi	3632360.64	0.22	3206790.90	68.93	**3053883.41**	0.32
u-i-hilo	82413.30	0.24	76969.72	66.70	**75594.93**	0.33
u-i-lohi	122044.94	0.24	108747.50	68.54	**103355.06**	0.32
u-i-lolo	2777.16	0.25	2599.61	66.07	**2552.42**	0.32
u-s-hihi	4897763.92	0.19	4436362.83	64.73	**4255904.34**	0.34
u-s-hilo	105157.39	0.18	99117.78	64.25	**97587.99**	0.33
u-s-lohi	163927.91	0.16	149018.93	63.17	**142941.47**	0.35
u-s-lolo	3527.45	0.17	3323.14	63.03	**3275.20**	0.33

Table 2: Results of running the local search on the solutions found by Min-min compared with the results from (Braun *et al.* 2001) for Min-min and the GA. The best result for each problem is shown in bold.

longer to locally optimise poor solutions such as those generated at random.

Dynamic scheduling

This section provides results for applying the local search procedure to the solutions found by both Min-min and a re-implementation of the SH algorithm described in (Maheswaran *et al.* 1999). The complex event simulator used in (Maheswaran *et al.* 1999) was not available for these experiments, however because we are simply interested in establishing if the local search procedure can improve solutions found by SH, the same ETC matrices used in the static problem can be used here for comparison. The SH implementation tested 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 reducing the number of jobs to be scheduled we give it a more similar problem to that used in the dynamic environment simulator. Experiments using only 64 jobs (and 16 processors as before) show that SH does often outperform Min-min on smaller problems.

The makespans found and time taken by Min-min, Min-min plus local search, SH and SH plus local search for scheduling the first 64 jobs from the first problem (i.e. where n is 0) in each ETC matrix class are shown in table 3. (All 100 instances in each class were not used as these results are merely intended to show the effect of applying the local search procedure to Min-min and SH, and so provide motivation to testing it in a dynamic environment. A full test run is therefore unnecessary.)

These results are not directly comparable with those provided in (Maheswaran *et al.* 1999), but they do indicate that the local search can effectively improve solutions found by both the SH and Min-min algorithms, and can do so sufficiently quickly that it could be used effectively in conjunction with such construction heuristics in a batch mode dynamic scheduling system.

It is interesting to note that although SH generally outperforms Min-min (except in two cases, *u-c-lohi.0* and *u-i-lolo.0*), after local search is applied to their initial solutions SH only outperforms Min-min on 7 of the 12 problems. Another interesting result is that for some problems, such as *u-s-lohi.0*, SH finds a shorter initial makespan than Min-min, but the local search procedure can improve the Min-min solution more than it can the SH solution. These results are therefore rather inconclusive as to which is the best overall approach to use. Perhaps, given the very short running times, both approaches could be tested and the best overall solution used.

These results are promising, but before any useful conclusions can be drawn more experimentation is required to find out if the local search can work as well when used in a more realistic dynamic environment (such as the simulator used in (Maheswaran *et al.* 1999)).

Conclusions

Scheduling independent jobs in heterogeneous computing environments is useful for several different considerations in domains such as grid computing. The local search procedure described here can improve solutions found by both static and dynamic scheduling heuristics, finding the best known makespans for some benchmark problems. The local search procedure is very simple, but experimentation shows that it can significantly improve solutions built by several construction heuristics.

Further work could experiment with alternative local search techniques, such as using different solution neighbourhood definitions and altering the method used to select a new solution. A tabu search (Glover & Laguna 1997) approach, using a similar neighbourhood definition to that used here, could also help to overcome local optima in the search space and further improve solutions. Moreover, the results provided here suggest that future researchers looking at entirely new solution methods for such problems should consider combining them with local search techniques, as they can significantly improve solutions for comparatively little effort.

Acknowledgments

We would like to thank Tracy Braun and Howard Siegel for sharing their test data and detailed results with us.

	Min-min		Min-min+LS		SH		SH+LS	
problem	makespan	time	makespan	time	makespan	time	makespan	time
u-c-hihi.0	1760833.29	0.00	**1281095.56**	0.06	1646667.56	0.05	1461943.25	0.05
u-c-hilo.0	28912.38	0.00	24429.20	0.00	27062.69	0.05	**23091.48**	0.05
u-c-lohi.0	57232.30	0.06	**42061.37**	0.06	62706.13	0.05	52613.48	0.05
u-c-lolo.0	998.92	0.05	771.26	0.05	880.11	0.00	**770.37**	0.06
u-i-hihi.0	703868.21	0.05	**565575.25**	0.05	619191.40	0.05	**565575.25**	0.00
u-i-hilo.0	12176.13	0.00	10298.82	0.00	10527.72	0.00	**9610.98**	0.00
u-i-lohi.0	23604.18	0.05	**19683.95**	0.05	21160.64	0.00	20844.77	0.00
u-i-lolo.0	493.57	0.06	445.01	0.06	541.95	0.00	**418.61**	0.00
u-s-hihi.0	1190978.03	0.05	872214.06	0.05	1090510.22	0.00	**848478.41**	0.00
u-s-hilo.0	21627.07	0.00	15529.14	0.00	15935.55	0.00	**15027.31**	0.00
u-s-lohi.0	30529.32	0.06	**19808.53**	0.06	28839.41	0.00	24003.64	0.00
u-s-lolo.0	925.05	0.06	616.20	0.06	792.78	0.05	**599.37**	0.05

Table 3: Results of running the local search 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.

References

Alvim, A. C. F.; Glover, F.; Ribiero, C. C.; and Aloise, D. J. 1999. Local search for the bin packing problem. available from: http://citeseer.ist.psu.edu/alvim99local.html.

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.

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

Glover, F., and Laguna, M. 1997. *Tabu Search*. Boston: Kluwer Academic publishers.

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, 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.

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