Instant-access cycle-stealing for parallel applications requiring interactive response

P.H.J. Kelly, S. Pelagatti* and M. Rossiter**
Department of Computing, Imperial College, London SW7 2BZ, UK
{p.kelly , s.pelagatti}@doc.ic.ac.uk

Abstract. In this paper we study the use of idle cycles in a network of desktop workstations under unfavourable conditions: we aim to use idle cycles to improve the responsiveness of interactive applications through parallelism. Unlike much prior work in the area, our focus is on response time, not throughput, and short jobs - of the order of a few seconds. We therefore assume a high level of primary activity by the desktop workstations' users, and aim to keep interference with their work within reasonable limits.
We present a fault-tolerant, low-administration service for identifying idle machines, which can usually assign a group of processors to a task in less than 200ms. Unusually, the system has no job queue: each job is started immediately with the resources which are predicted to be available.
Using trace-driven simulation we study allocation policy for a stream of parallel jobs. Results show that even under heavy load it is possible to accommodate multiple concurrent guest jobs and obtain good speedup with very small disruption of host applications.
Keywords: parallel computing, cycle stealing, performance prediction, distributed computing

1 Introduction

This paper concerns the feasibility of on-the-fly recruitment of idle workstations to enable parallel execution of short computationally-intensive phases of an interactive application, as commonly arise in a computer-aided design environment. In such applications, when the user is constructing the design, little processing power is required, however when the user selects 'Generate Photo-realistic Image', the computation required increases dramatically. Ideally, the user would not want to wait long for the image to be produced, possibly grabbing spare processing time from unused workstations.

Our objective is to exploit the fact that (as we quantify below) even when a machine is actually being used interactively (the "host" job), there are often periods of inactivity lasting several seconds or more. We focus on the challenging goal of using these brief periods of idleness to execute short "guest" jobs in parallel in order to enhance response time.

In addition to presenting a simple and effective software tool, we explore the potential for achieving this objective. We have chosen an extremely difficult environment - a heavily-used student laboratory of 32 Linux PCs; see Figure 1. We show that a typical (albeit rather simple) parallel task can reliably achieve a speedup of 3 or more (reducing runtime to ca.14 seconds), while interfering with only 6-7% of host user seconds. Furthermore, we evaluate a simple allocation policy which handles intermittent arrival of such tasks.

Cycle stealing on networks of desktop workstations The idea of making use of this wasted processing power is attractive and exploiting idle workstations has been a popular research area. Studies have shown that in typical networks of workstations (NOWs), most machines are idle most of the time [2, 1]. Batch systems like Condor [9] have been in use for years to utilize idle workstations for running independent sequential jobs. There have also been studies on the possibility of using idle workstations for parallel processing on coarse grain parallel jobs. Arpaci *et al.* [2] study the availability traces of a 60-workstation pool using a job arrival trace for a 32 node CM-5 partition. They find that the pool is able to sustain the 32-node parallel workload in addition to the sequential load imposed by interactive tasks. Similarly, Acharya *et al.* [1] show that for three non-dedicated pools of workstations it was possible to achieve a performance equal to that of a dedicated parallel machine between one third and three quarters the size of the pool. The

* On leave from Dipartimento di Informatica, Corso Italia, 40, 56125 Pisa, Italy. e-mail: susanna@di.unipi.it.
** Now with Telcordia, Inc, New Jersey, USA

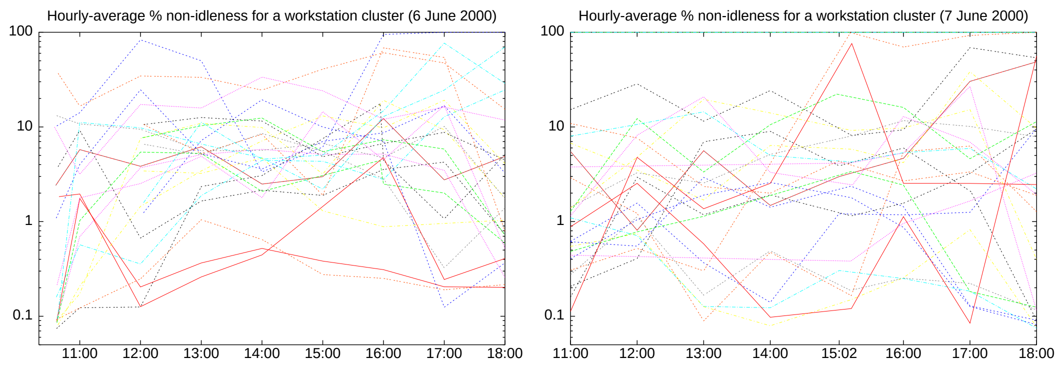


Fig. 1. This graph shows (on a log scale), the hourly-averaged percentage utilisation (see Section 3.1) of our 32 Linux PCs over two typical days. Although not always 100% busy, the machines are essentially in continuous use.

results were achieved on relatively coarse grain *adaptive* parallel applications which could dynamically reconfigure to cope with changes in the pool of idle workstations available.

Instant-access cycle-stealing for interactive response In contrast to this earlier work, we focus on interactive applications with intermittent bursts of computation load. This requires optimizing the average response time for individual guest jobs and not the global system throughput. Second, our computation bursts are quite short (10-20 seconds if executed in parallel). This rules out the possibility of expensive process migrations during computation and makes crucial the ability to foresee idle times accurately. It is impractical to ship code and data to distant specialized nodes as happens in grid-oriented metacomputing environments [4, 7, 12]. Finally, since the guest jobs arise from interactive applications, we have to exploit idle workstations during busy day hours and we are not interested in patterns of idleness during nights or weekends.

To our knowledge, this is the first attempt to investigate idle workstation harvesting in this particular setting. There are two, linked challenges:

1. Can we achieve a useful speedup? Parallel programs (especially short-running ones) rely on all processors making progress. If just one of the participating machines is poorly-chosen, the entire parallel task will be delayed.
2. Is the interference with the host machines' other user(s) excessive?

Contributions The main contributions of this paper are:

1. We present a low-overhead distributed recruitment service, which automatically identifies the available workstations on a local-area network. By autonomously electing a leader, the service requires minimal administration and handles failures gracefully.
2. We analyse traces of workstation utilisation, in order to quantify the idle time available on a network of workstations, its predictability, and the potential for using idle time for parallel processing.
3. Using a simulation driven by these traces, we evaluate the guest job performance achievable, and the amount of interference to host jobs.
4. We investigate scheduling policies to deal with a workload consisting of multiple users generating occasional computationally-intensive guest jobs.

The paper is organized as follows. Section 2 gives an overview of the architecture of the system proposed, Section 3 reports on the experimental results obtained, Section 4 discusses some related work and Section 5 concludes.

2 System overview

The system is organized as a network of daemon processes, one for each workstation. Daemons monitor local load, provide job startup services and cooperate to predict future load and to schedule incoming guest

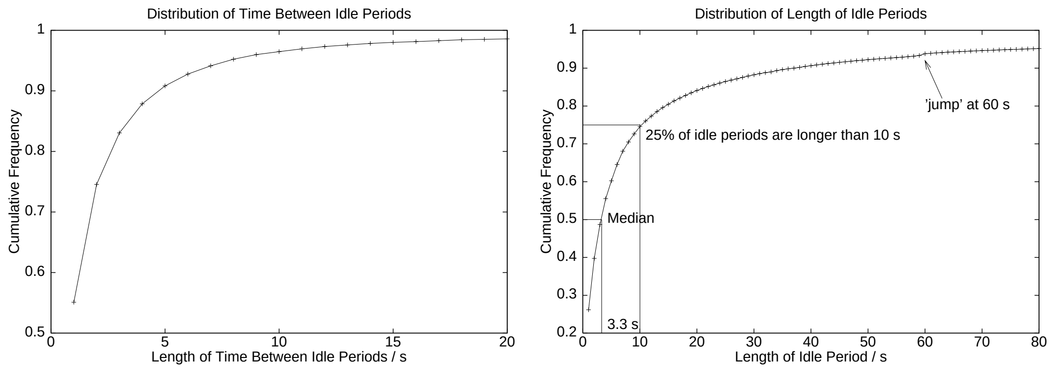


Fig. 2. Distribution of time between idle periods (left) and distribution of length of idle periods (right).

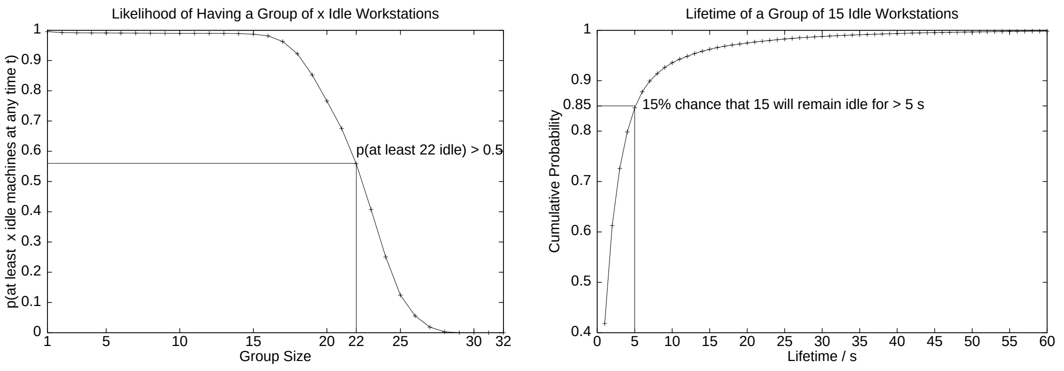


Fig. 3. Likelihood of having x idle workstations at the same time (left) and expected lifetime of a group of 15 idle workstations (right).

jobs. As the guest jobs are fairly short (10-20 seconds), we restrict our attention to clients and servers on a single LAN running under the same administration/domain.

The mpidled monitor process Allocation is orchestrated by a *leader* daemon which acts as a central server. The leader is elected using the distributed protocol by Garcia-Molina [8]. The protocol ensures automatic substitution of a leader in case of suspected failure. When a client wants to spawn a new guest job it makes a recruitment request to the leader which, after querying the daemon processes, returns a list of machines predicted to be idle for the near future. Then, the client can contact the daemon on each machine to inform it of the program to be executed. Each daemon process is responsible for monitoring the system status and computing a load prediction (Section—3.1). The leader, which may be any one of the daemons, is responsible for allocating resources (Section—3.2).

The mpidle application and API A client can initiate a request for resources using a command line utility (`mpidle`) which produces a list of idle workstations, as a parameter of an MPI job. Alternatively, a lower-overhead API is provided for direct invocation from within client applications.

3 Experimental evaluation

Overview Section 3.1 quantifies the amount of idle time likely to be found in a typical LAN environment during the day. Section 3.2 discusses and evaluates our load prediction strategy. Section 3.3 evaluates the time spent in finding a suitable workstation pool to execute new guest jobs (*recruitment overhead*). Section 3.4 presents the simulation results under various scheduling strategies.

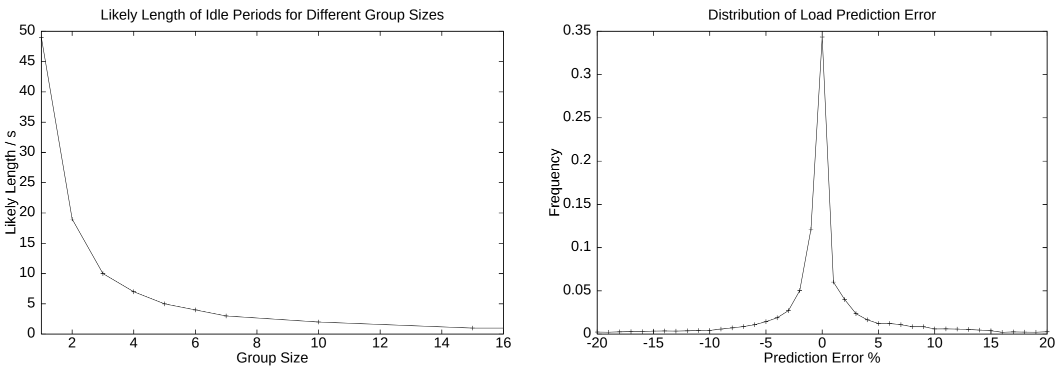


Fig. 4. Likely length of idle periods for different group sizes (left) and distribution of load prediction errors (right).

3.1 Idle workstation recruitment

We consider a workstation *idle* if it is not executing user processes and has a significant amount of spare CPU time. More precisely we define a workstation as *idle* if, over a one second period, less than 10% of CPU time is spent executing user processes, and at least 90% of CPU time could be devoted to a new process.

Experimental environment To measure idleness patterns using our recruitment policy we carried out observations of load traces collected over two weeks on a pool of 32 very similar non-dedicated workstations (300MHz and 350MHz Pentium II, 128MB, Redhat Linux 6.1) located at Imperial College London. This is a uniform pool of publicly available machines used fairly intensively by undergraduate computer science students for course assignments, software development projects, web browsing and email. Traces were collected during the busy daytime hours, weekdays 9am to 6pm.

Pattern of workstation utilisation Of all the one-second samples, 86% were idle. Idle periods occur very frequently. Figure 2.*left* shows the distribution of time between idle periods – 55% of intervals are $1s$ or less. Figure 2.*right* shows the distribution of length of idle periods over all workstations. 50% of idle periods last for at least $3.3s$. One quarter of all idle periods last for longer than $10s$: idle workstations often remain idle long enough to perform another useful task. (Note the small inflection in the plot at $60s$, indicating that there are occasionally 'periodic' processes running on the workstations that cut-short idle periods that would have otherwise exceeded $60s$.).

To evaluate scope for parallel guest jobs, we studied the patterns of idleness across *groups* of workstations. Figure 3.*left* shows the probability of having a group of workstations of a given size at any given time. A group of 15 idle workstations is very nearly guaranteed to be available at any time, and a group of 22 is available with a rather high probability. The stability of such groups is shown in Figure 3.*right*. A group of 15 idle workstations is unlikely to remain idle for very long - there is only a 15% chance of them lasting for more than 5 seconds. Smaller groups are normally more robust (Figure 4.*left*).

3.2 Predicting short term workstation load

Each daemon process monitors its CPU load once every second. When an availability request is received from the leader a load prediction is computed and returned. Load is predicted using a windowed mean of recent load measurements to predict the load over the next few seconds. Previous studies [5, 14] have shown that accurate short-term load prediction is possible and that good predictions can be made simply by taking the mean of recent load measurements. However, the load metric considered in [5, 14] (UNIX 'Load Average' - the average length of the run-queue) is different from the metric being considered here (CPU activity) and so we evaluated the accuracy of their prediction scheme with our metric. The windowed-mean prediction scheme was applied to our load traces, and the prediction errors were computed. Figure 4.*right* shows that the error obtained using a window of 5 measurements is usually very small - 35% of predictions correctly forecast the average load over the following $10s$. We also studied the relationship between the

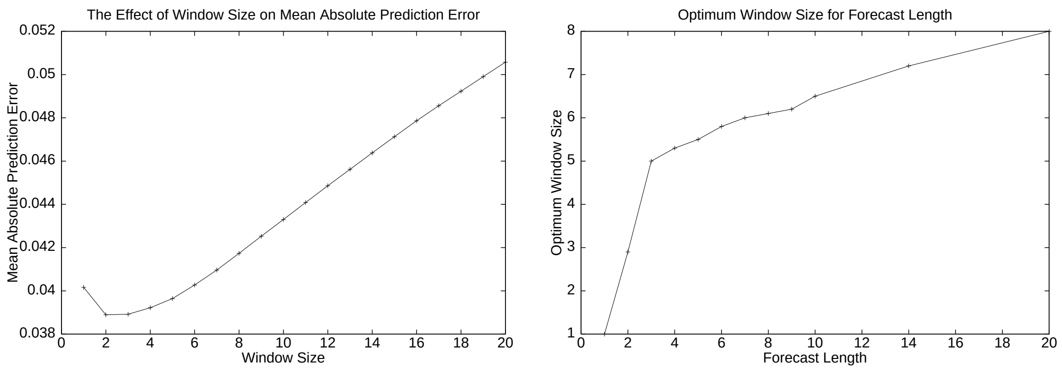


Fig. 5. Effect of window size on mean absolute error (left) and optimum window size for forecast length (right).

window size and the length of the period for which the prediction is needed. Figure 5.*left* shows the effects of window length on the mean absolute error for a particular desired prediction length (in this case $10s$). Figure 5.*right* shows optimal window sizes for different forecast length periods.

3.3 Recruitment overhead

The *workstation recruitment overhead* is the time spent in finding a suitable workstation pool to execute a new guest job. Figure 6.*left* shows the measured recruitment time during our experiment. The vast majority of recruitment requests are answered within a very short time ($\leq 0.15s$), however a small number of requests can be delayed to anything up to $2.5s$. This happens when requests occur when the leader is executing a periodic check to ensure that there is no other leader in the cluster. During this time it cannot claim to be the leader and any request must wait until the periodic check is finished [8].

3.4 Evaluating scheduling strategies

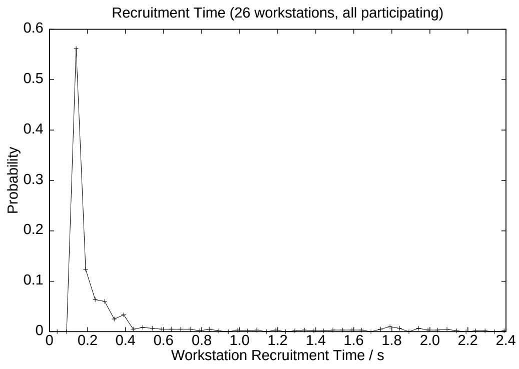
Trace-driven simulation To ensure reproducibility of results and allow for closer insight of the system behavior, we constructed a simulation using the load traces discussed in Section 3.1, varying various parameters. We tested the system with a sample rendering application which takes $42s$ on a single workstation. Figure 6 shows its speedup behavior when executed on a dedicated cluster of the workstations.

The simulation uses the application's speedup curve to predict the expected completion time of each task on the resources available. It also accounts for the delay incurred (to all participating processors) when a guest process contends with a host process for CPU time. The contention which occurs is determined from the load traces, which record the number of running processes during each second so that a process's CPU time share can be computed.

The simulated usage regime To exercise the resource allocation mechanism, we simulate a fairly intensive situation in which clients request execution of rendering jobs at random intervals. The rendering jobs are all of the same size ($42s$ on one processor). Requests arrive with an exponential distribution, with a mean inter-arrival time of $20s$.

Scheduling strategies We experimented with three different scheduling strategies: **random**, **no reserve** and x**-reserve**. The results are shown in Table 1. For each scheduling strategy we measured the following:

- *Jobs Refused* the proportion of submitted guest jobs for which there were no available participants;
- *Idle Seconds Used* the proportion of idle seconds in the day that were put to good use by the system;
- *Mean Group Size* the mean size of the group of workstations allocated to incoming guest jobs;
- *Mean Speedup* the mean speedup for guest jobs including those for which no workstations were available (i.e. those that were forced to execute sequentially);
- *Seconds Disrupted* the proportion of busy seconds that were disrupted by the execution of guest jobs, i.e. how often did a misprediction lead to disruption of ordinary workstation applications.



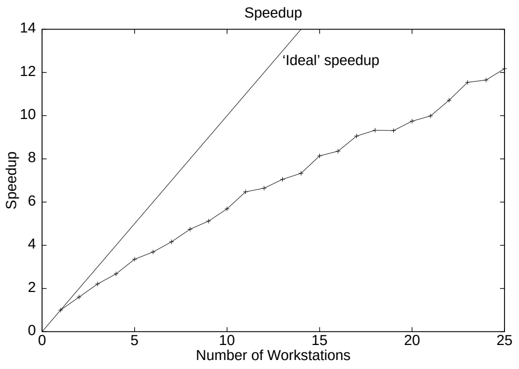


Fig. 6. Recruitment overhead on the sample application(left) and speedup on a dedicated workstation network(right).

Scheduling strategy	**rand**	**no res**	**10-res**	**20-res**	**30-res**	**40-res**	**50-res**	**60-res**
Jobs Refused	nil	16.3%	3.3%	1.5%	2.0%	0.5%	0.2%	0.1
Idle Seconds Used	25.0%	21.6%	23.3%	23.2%	22.6%	22.1%	21.5%	21.0%
Mean Group Size	17.0	17.2	15.24	13.6	12.0	10.3	8.7	7.1
Mean Speedup	3.68	3.58	4.58	4.88	4.96	4.82	4.46	3.92
Seconds Disrupted	44.4%	5.28%	6.3%	6.5%	5.9%	6.0%	5.9%	6.2%

Table 1. System behavior results for different scheduling strategies.

The random policy We show the performance of a random allocation policy as a control experiment. A constant number of workstations is recruited for every job and this set is chosen at random among all the workstations regardless to their load. We used a constant group size of 17, which is near to the mean which results under the no-reserve policy.

The "no-reserve" policy The no-reserve policy allocates all the idle workstations available to each recruitment request. Should a second request arrive shortly afterwards, no idle workstations will be left.

– This led to a slightly worse speedup than random allocation (3.58).
– However, a large proportion (84%) of recruitment requests were satisfied.
– 20% idle seconds were exploited, out of the average 25% of seconds belonging to periods of at least 10 seconds. This could be improved, especially since jobs were refused.
– The proportion of seconds that were disrupted by inappropriate allocation of jobs was low (5.3%), although not low enough for the system to be considered completely non-intrusive.

The x-reserve policies The x-reserve strategies try to save x% of the resources available at any given time for (near) future requests in order to have a better distribution of the group sizes and to lower the percentages of guest jobs refused. With no-reserve, a large proportion of jobs were executed on small numbers of workstations or were forced to be executed serially because no workstation was available. Table 1 shows the results obtained with x-reserve strategies keeping a different proportion x of reserve at each allocation (the no-reserve strategy is the same as the 0-reserve strategy). The results are as follows:

– By choosing the right reserve percentage we can achieve an average speedup of up to 4.96.
– Furthermore, this increase in speedup is achieved without significantly increasing the proportion of the seconds disrupted.
– The speedup falls when too many workstations are kept in reserve as the average group size drops.
– Keeping reserves reduces the percentage of jobs refused, reducing the variance of the speedup experienced by different guest jobs.

The effect of the reserve percentage on the distribution of group sizes is illustrated in Figure 7. As expected, for small reserves, groups are either very large or very small, while for larger reserves the group sizes are close to the mean.

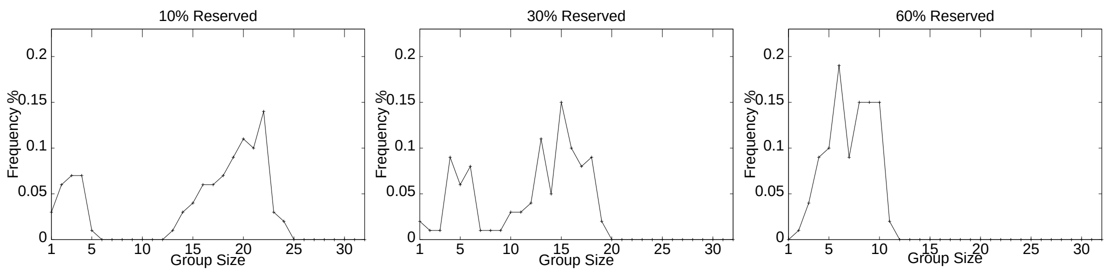


Fig. 7. The effect of x-reserve scheduling policies on distribution of group sizes.

4 Related work

With Condor [9], the aim is to speed up independent sequential guest jobs using idle workstations in a LAN. Usually, the jobs require a large amount of computation (hours more than seconds) and a network of monitor daemons is used to collect information on the current load of machines on the net. Disruption of host jobs is minimised by migrating the guest job as soon as the host jobs need a workstation. Linger-Longer [11] works in the same scenario but allows a guest job to remain on a host machine when it ceases to be idle. To avoid disruption it employs a set of Linux kernel extensions which use a new guest process priority to prevent guest processes from stealing time from host processes and a new page replacement policy which limits the slow down caused by guest pages in the virtual page system. With this new scheme, the authors claim a much effective usage of workstations, allowing gains up to 60% in the total compute time with respect to Condor. Although the techniques used in these systems can be used in our setting, the focus of our work is on *interactive* parallel guest jobs posing a quite different set of challenges.

As mentioned in the Introduction, the use of idle workstations to execute a batch queue of parallel jobs has been studied by Acharya *et al.* and Arpaci *et al.* [1, 2]. With the longer-running jobs they study, processes can be migrated from machine to machine during execution. Furthermore, their objective was to minimize the execution time of a whole batch, which can mean very long execution time for single jobs in order to achieve better global resource arrangement.

Some of the problems addressed in our research, such as workstation load prediction and load sensitive guest job scheduling have been addressed recently in the broader framework of WAN scale metacomputing systems [7, 4, 12]. This setting is much more complex than ours and requires network load prediction to be addressed. Moreover, the higher overhead due to non local job scheduling is more suitable for coarser grain guest jobs than the ones addressed in this study.

Finally, scheduling parallel computations on batch parallel systems has attracted considerable attention [3, 6, 13, 10]. The usual metric to be optimized here is global batch throughput. However, Subholk *at al.* [13] proposes strategies to minimize response time for individual applications. They take both communication load and computation load into account and select a pool of workstation and communication links to be used. Our research addresses LAN environments in which only computational load is relevant for node selection. The strategy proposed by Subholk *et al.* for our specific problem corresponds to our no-reserve policy. As we discussed in Section 3 this strategy penalizes future jobs and leads to smaller average speedup figure with respect to x-reserve. Although more experiments are needed, we believe that, in our setting, a strategy aiming to optimize the average speedup experienced by competing guest jobs leads to better resource usage and more reliable behavior than optimizing the response time of a guest job in isolation.

5 Conclusions and directions for further research

We have provided evidence that interactive performance of applications with intermittent computational demands can be substantially enhanced through opportunistic parallel execution on other instantaneously-idle workstations on the same LAN. Some interference with host tasks is incurred, but the effect is small. When guest job requests arrive frequently, much better performance is achieved by holding back some of the available resource on each allocation.

While there is enormous scope for further work, this paper has demonstrated “mpidled” to be a simple yet surprisingly effective tool. The software is in regular use at Imperial College and a public release is planned. Further research is needed:

– How would our results change with different levels of host load? We have taken a fairly extreme situation of essentially continuous utilisation - many realistic environments would give better results.
– Our simple policy of holding back some resources for future requests appears fairly stable, but we would like to characterise how the policy should be adjusted as task arrival rate and host load are varied. Some kind of adaptive scheme looks attractive.
– Our definition of “idle” is somewhat arbitrary (Section 3.1). We need to evaluate how lowering the idleness threshold would reduce interference, and reduce speedup. In our environment, external users (and Windows users) often connect to our Linux systems remotely, so some level of interference to desktop responsiveness is already tolerated. Other organisations have a different culture.
– We used a rather simple parallel application to exercise the system. Although our rendering application has less-than ideal speedup, it is relatively loosely-synchronised. We have been using mpidled to run a tightly-synchronised CFD solver and have positive practical experience but have not yet been able to quantify the resulting performance.
– Realistic applications often (like the CFD solver) have large input and output files. This is easily addressed by using the local filesystem on the machines allocated by mpidled - but the next interactive use of the application (which uses the results from the previous run) is likely to be allocated a differing set of machines. We plan to explore strategies for achieving parallel file access while retaining the necessary scheduling flexibility.

Acknowledgements This work was partially supported by an EPSRC Visiting Fellowship (GR/N63154).

References

1. A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computing. In *Proc. 1997 ACM SIGMETRICS Intl. Conf. on Measurement & Modeling of Computer Systems*, pages 225–236.
2. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. A. Anderson, and D. A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proc. 1995 ACM SIGMETRICS Intl. Conf. on Measurement & Modeling of Computer Systems*, pages 267–278, Ottawa, May 1995.
3. M. J. Atallah, C. L. Black, D. C. Marinescu, H. J. Siegel, and T. L. Casavant. Models and algorithms for co-scheduling compute-intensive tasks on networks of workstations. *J. Parallel & Distr. Computing*, 16:319–327, 1992.
4. H. Casanova and J. Dongarra. Netsolve: A network server for solving computational science problems. *Intl. J. Supercomputer Applications & High Performance Computing*, 11(3):212–223, 1997.
5. P. Dinda and D. O’Hallaron. An evaluation of linear models for host load prediction. In *Proc. 8th IEEE Symp. on High-Performance Distributed Computing (HPDC-8)*, Redondo Beach, CA, Aug. 1999.
6. A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. In *Proc. 1996 ACM SIGMETRICS Intl. Conf. on Measurement & Modeling of Computer Systems*, Philadelphia, PA, 1996.
7. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.
8. H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comp.*, C-31(1):47–59, Jan. 1982.
9. M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th Intl. Conf. of Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988. IEEE-CS Press.
10. F. Petrini and W. Feng. Buffered coscheduling: a new methodology for multitasking parallel jobs on distributed systems. In *Proc. Intl. Parallel & Distributed Processing Symp. 2000*, Cancun, MX, May 2000.
11. K. D. Ryu and J. K. Hollingsworth. Linger-Longer : Fine-grain cycle stealing for networks of workstations. In *Proc. Supercomputing’98*, Orlando, Nov. 1998.
12. S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network based information library for globally high performance computing. In *Proc. Parallel Object-Oriented Methods & Applications (POOMA)*, Santa Fe, New Mexico, Feb. 1996.
13. J. Subholk, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on the networks. In *Proc. 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP’99)*.
14. R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared Unix systems. In *Proc. 8th IEEE High Performance Distributed Computing Conf. (HPDC8)*, Redondo Beach, CA, Aug. 1999.