



THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-04-01

Local Scheduling out-performs Gang Scheduling on a Beowulf Cluster

Peter Strazdins and John Uhlmann

Jan 2004

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

<http://cs.anu.edu.au/techreports/>

Recent reports in this series:

- TR-CS-03-02 Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. *A garbage collection design and bakeoff in JMTk: An efficient extensible Java memory management toolkit*. September 2003.
- TR-CS-03-01 Thomas A. O'Callaghan, James Pople, and Eric McCreath. *Building and testing the SHYSTER-MYCIN hybrid legal expert system*. May 2003.
- TR-CS-02-06 Stephen M Blackburn and Kathryn S McKinley. *Fast garbage collection without a long wait*. November 2002.
- TR-CS-02-05 Peter Christen and Tim Churches. *Febri - freely extensible biomedical record linkage*. October 2002.
- TR-CS-02-04 John N. Zigman and Ramesh Sankaranarayana. *dJVM - a distributed JVM on a cluster*. September 2002.
- TR-CS-02-03 Adam Czezowski and Peter Christen. *How fast is -fast? Performance analysis of KDD applications using hardware performance counters on UltraSPARC-III*. September 2002.

Local Scheduling out-performs Gang Scheduling on a Beowulf Cluster

Peter Strazdins and John Uhlmann,
Department of Computer Science, Australian National University
Acton ACT 0200 AUSTRALIA
{Peter.Strazdins@cs.anu.edu.au,John.Uhlmann@anu.edu.au}

Abstract

Gang Scheduling and related techniques are widely believed to be necessary for efficient job scheduling on distributed memory parallel computers. This is because they minimize context switching overheads and permit the parallel job currently running to progress at the fastest possible rate.

However, in the case of cluster computers, and particularly those with COTS networks, these benefits can be overwhelmed in the multiple job time-sharing context by the loss the ability to utilize the CPU for other jobs when the current job is waiting for messages.

Experiments on a Linux Beowulf cluster with 100 Mb fast Ethernet switches are made comparing the SCore buddy-based gang scheduling with local scheduling (provided by the Linux 2.4 kernel with MPI implemented over TCP/IP). Results for communication-intensive numerical applications on 16 nodes reveal that gang scheduling results in ‘slowdowns’ up to a factor of two greater for 8 simultaneous jobs. This phenomenon is not due to any deficiencies in SCore but due to the relative costs of context switching versus message overhead, and we expect similar results will hold for any gang scheduling implementation.

A performance analysis of local scheduling indicates that cache pollution due to context switching is more significant than the direct context switching overhead on the applications studied. When this is taken into account, local scheduling behaviour comes close to achieving ideal slowdowns for finer-grained computations such as Linpack. The performance models also indicate that similar trends are to be expected for clusters with faster networks.

Keywords: parallel computing, job scheduling, gang scheduling, cluster computing

1 Introduction

Gang scheduling [6], or one of its many variants [11, 16, 19], is widely believed to be necessary for optimal distributed memory multiprocessor resource utilization with time-shared jobs with medium- to fine-grained communication patterns.

These techniques require that, on each node, the processes related to a particular parallel task (*gangs*) are scheduled to run simultaneously. Gangs are packed together and are preempted and rescheduled as a unit. As well as reducing context switching overheads, this ensures messages associated with a parallel job are received as soon as possible after delivery, and hence permits that parallel job to progress optimally.

However, the studies that established the above belief by making direct comparisons seem to be very few. The only one known to us that makes a direct evaluation of gang scheduling (compared with local scheduling) is [6]; this uses a synthetic program (which repeatedly executes a computation loop followed by a barrier) on an early NUMA shared memory architecture. But shared memory parallel processors and older-style Massively Parallel Processors (MPPs) have relatively lower communication overheads than for modern clusters. Furthermore, the assumption underlying gang scheduling: that any messages received are destined for the currently executing process, simplifies (and possibly speeds up) the implementation of communication [16].

In the only survey of job scheduling on clusters known to the authors, it is claimed that “*Scheduling on [clusters] is essentially the same as on commercial MPP systems*” [4]. Clusters however are significantly different from traditional MPPs: they have a slower network, and normally have a standard OS residing on each node. Furthermore, cluster communication can be thought of as network I/O; typically applications will block when waiting on messages.

Explicit co-scheduling normally requires considerable infrastructure to be added to cluster. Existing systems include SCore [10] and ParPar [9].

An alternative is *implicit co-scheduling*, where the processes in the gang are not explicitly coscheduled but should coschedule themselves through their behaviour and local scheduling policies, e.g. if a message arrives for a non-current job, a CPU might preempt the current job and schedule the new one [12, 3]. These are potentially less complex to implement. In the context of more coarse-grained jobs (with infrequent communication phases) on clusters, it has been suggested that co-scheduling can occur without any measures being taken, simply due to the fact that processes get higher priority when communicating [4].

However, the simplest technique of all to implement is (uncoordinated) *local scheduling*, where no measures are taken to coschedule processes in the gang. In this case, gangs may not be coscheduled at all. This would be the default scheduling policy on a raw Linux cluster. On such a cluster at our university [2], our experience has been that when $j \geq 2$ competing medium- to fine-grained jobs ran on the cluster, the resulting performance degradation (compared with when each job ran by itself) was much greater than a factor of j . Thus, we desired to install a gang scheduling system on our cluster and see to what extent it would alleviate this problem [17].

A recent paper [18] introduced *paired gang scheduling*, in which the processes of two jobs are regarded as being in the same gang, and the local scheduler on each CPU chooses (the process corresponding to) the job to be run at any given time. While primarily motivated to improve CPU utilization when some of the processes are blocked for I/O, paired gang scheduling significantly out-performed gang scheduling for a dynamic workload on the ParPar cluster, even when the only I/O was that due to messaging. Here, a synthetic program (with negligible memory footprint) similar to that in [6] was used to create the workload. The length of the compute loop was adjusted so that the CPU utilization for a single such job was 45% [18]. While the authors concluded that “*paired gang scheduling seems to be a good compromise between the extreme alternatives of gang scheduling and uncoordinated local scheduling*” [18], no comparisons to local scheduling were

made.

The main original contributions of this paper are as follows. It gives a direct comparison of gang and local scheduling on clusters. It also uses real applications to make this comparison, and demonstrates why the use of these is important in evaluating scheduling policies. Conclusions that can be drawn from the results of the paper do not seem to be reflected in the current literature. Finally, it also provides a performance analysis for local scheduling, and proposes and evaluates an optimistic performance model for it.

This paper is organized as follows. Section 2 describes the Bunyip Beowulf cluster used in our experiments, and Section 3 describes the SCore cluster management system. Our experimental setup and a comparison of gang and local scheduling is given using (non-synthetic) parallel applications in Section 4. The performance of local scheduling is further analysed in Section 5. Conclusions are given in Section 6.

2 The Bunyip Beowulf Cluster

The Beowulf cluster *Bunyip* [2] used in our experiments is a 96 node dual processor Pentium III's running at 550 MHz. Bunyip runs a Linux 2.4.18 kernel. Each CPU in a node has a non-shared 256 KB direct-mapped second-level cache.

Bunyip consists of 4 groups made up of 24 nodes each. The nodes each has three 100 Megabit NICs, each being connected to an intergroup switch Packard ProCurve 4000M); thus every node is directly connected to every other through one of these switches.

Normally, two processes of a parallel job are spawned on each node; ideally, each will be scheduled to one of the node's two CPUs at the same time.

2.1 Multiple Jobs under Linux and LAM MPI

LAM MPI 6.3.2 [15] is the version of MPI normally used by parallel programs on the Bunyip. Point-to-point message send and receive uses a TCP/IP transport, unless the MPI processes are on the same node (in which case, it is performed by a `sysv` interprocess communication transport, which uses shared memory).

Under the TCP/IP transport, a 'small' (in this case $\leq n_s = 64\text{KB}$) message is effectively non-blocking to the sender, and as soon as the message is buffered (ready to send), execution of the calling application process resumes [15]. If the message is larger than n_s , it is broken down into packets of size n_s , with an acknowledgement for the receiver required before each subsequent packet is sent.

A message receive results in a `recvfrom()` system call for each required packet. If an appropriate packet has already arrived at the time of the call, the payload is copied into the user space receive buffer, and execution of the calling (application) process resumes. Otherwise, the calling process *yields*, and other processes may be scheduled by the kernel for execution. When an appropriate packet arrives, it generates an interrupt and the calling process will be scheduled as *runnable* by the kernel.

In the case of the `sysv` transport, if an appropriate packet has not yet arrived at the time of the call, it will *poll* for the packet; if it does not arrive in a certain interval, the calling process similarly yields.

Thus, if multiple MPI jobs are running over a subset of the nodes, when a process for one job is blocked waiting for a message, the node’s CPU can be utilized by running the process for another job. In other words, it is possible in principle to overlap the communication of one parallel job with the computation of another.

2.2 Message Performance under LAM MPI

Using a ping-pong benchmark running under LAM MPI’s `mpirun -c2c -O`, the time to send and receive a message of n doubles given by $t(n) = \alpha + \beta n$ where under the TCP/IP transport:

$$\alpha = 80\mu s, \beta = .85\mu s \quad (1)$$

and under the `sysv` transport:

$$\alpha = 42\mu s, \beta = .10\mu s \quad (2)$$

To determine the Linux context switch overhead, α_c , we used a benchmark program which executed a (short) compute-loop followed by a process yield system call, repeated y times. Using $\alpha_c = \frac{t_2 - 2t_1}{2y}$, where t_j is defined in Section 4.1, yielded a value much smaller than α :

$$\alpha_c = 2.5\mu s \quad (3)$$

3 The SCore Cluster Management System

The SCore system is developed by the PC Cluster Consortium [10]. It is freely available (including source code) and is in wide use in cluster installations world-wide. It consists of middleware that runs on top of full Linux kernels. The job scheduling algorithm is based on Distributed Hierarchical Control [7], but lacks the actual distributed hierarchical control aspect. This leaves a buddy-based gang scheduler with centralised control [8].

The scheduling in SCore is done through the use of priority queues. Each queue can have memory, disk and time limits and can be scheduled in either a time-shared (gang scheduling with a buddy node allocation) or exclusive fashion. The nodes are time-shared in a coordinated fashion between the jobs with a global synchronisation and flushing of the network between time slices. In the case of dual nodes, two (consecutive) processes are allocated per node.

Similar to LAM MPI, parallel jobs are run through the SCore’s version of `mpirun` command which connects to a SCore daemon process.

SCore 5.4.0 [10] was installed on Bunyip; it includes an implementation of MPI based on MPICH 1.4.0. SCore includes a lightweight TCP/IP replacement called PM/Ethernet; this reduces internode communication latency, primarily due to the fact that the application busy-waits upon message receipt rather than yields (cf. Equation 1), with the communication cost per double word also reduced slightly:

$$\alpha = 70\mu s, \beta = .82\mu s \quad (4)$$

Like LAM MPI, SCore is able to communicate between processes on the same SMP node using shared memory.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Figure 1. SCore allocation of $p = 32$ processes to an 8×4 grid on a dual CPU cluster

4 Evaluation of Gang and Local Scheduling on the Bunyip

This section gives an evaluation of the two scheduling policies on the Bunyip cluster for two numerical benchmark programs, a matrix multiply program and Linpack benchmark. Preliminary versions of these results were reported in [17].

4.1 Experimental Setup

In these experiments, j identical parallel jobs are spawned over the same subset of nodes. They are spawned in the background, i.e. as simultaneously as possible, via a control program. CPU monitoring indicated that both policies time-shared the jobs fairly evenly, i.e. they gave fair individual service to each job. Further experiments demonstrating fairness are reported in [17].

Thus, packing issues are avoided in these experiments, with an identical number of processes allocated to each node in the subset.

For these experiments, dedicated use of the cluster was obtained.

Under these conditions, scheduling performance can be captured by taking t_j , the elapsed execution time for all j jobs, and the job slowdown metric [5] reduces to:

$$s_j = \frac{t_j}{t_1} \quad (5)$$

The memory footprint of each job was set to be $\approx 6\%$ of a Bunyip’s node’s available user memory; thus up to $j = 8$ jobs could run simultaneously on each CPU without danger of performance degradation due to swapping.

SCore was installed on 16 nodes of Bunyip (it requires modifications to the Linux kernel). With p parallel processes, the programs using a logical $P \times Q$ processes grid, where $p = PQ$ and, for p being a power of 2, either $P = 2Q$ or $P = Q$. With a row-major allocation of processes to nodes, SCore allocates two consecutive processes per node, as shown in Figure 1. This has the effect of effectively increasing vertical communication time (e.g. processes 0 and 1 compete with processes 8 and 9 for a single link between the same two nodes of the grid), and in some circumstances decreasing horizontal communication time (e.g. processes 0 and 1 can communicate via faster shared memory transport). While this complicates program behaviour, SCore forces such an allocation on a dual CPU system. LAM MPI options were chosen to match this allocation.

The Linpack program solved an $N \times N$ column-major dense linear system, using storage blocking with a block size of 60 [13]. Such a computation has a $\Theta(N \lg P)$ communication startup cost, $\Theta(\frac{N^2 \lg P}{Q})$ communication volume cost, and a $\Theta(\frac{N^3}{PQ})$ computation cost [13]. For up to moderate values of N , such a computation can therefore be regarded as fine-grained.

The matrix multiply program performed the computation $C \leftarrow A * A + C$, where A and C are $N \times N$ matrices. It has a $\Theta(\frac{N^2}{Q} + \frac{N^2}{P})$ communication volume cost and an $\Theta(\frac{N^3}{PQ})$ computation cost. The program divides the inner index into blocks of $k = 128$, and repeatedly broadcasts vertically (horizontally) the next $k \times N$ ($N \times k$) segment of A , and uses these to perform a rank- k update on the local portion of C . This program thus has a much coarser granularity, with distinct communication and computation phases.

Both programs randomly generated input matrices, and discarded their output matrix. Only process 0 produced any output messages, which included the elapsed time for the main computational phase.

4.2 Results

p	N	Single job			Simultaneous jobs			
		%CPU	%Mem	Time (s)	2	4	6	8
LAM – local scheduling								
2	1000	99	5.2	3.7	1.81	3.45	5.10	6.75
4	1500	85	6.2	6.7	1.80	3.41	5.05	6.82
8	2000	71	5.2	9.6	1.81	3.44	5.05	6.62
16	3000	71	6.2	14.8	1.85	3.56	5.18	6.88
32	4000	64	5.4	20.9	1.83	3.64	5.24	6.96
SCore – gang scheduling								
2	1000	100	5.1	5.0	1.80	3.52	5.28	7.02
4	1500	100	6.2	7.9	1.84	3.63	5.51	7.22
8	2000	100	5.6	10.3	1.89	3.70	5.57	7.42
16	3000	100	6.2	15.2	1.92	3.82	5.75	7.71
32	4000	100	5.6	20.4	1.95	3.88	5.84	7.88

Table 1. $N \times N$ Matrix multiplication slowdowns, with p processes on p CPUs.

Tables 1 and 2 show single job performance and slowdowns per multiple jobs, with (near-) constant memory per CPU for each job. All results were run with p processes on a logical $P \times Q$ grid. These were run on $\frac{P}{2}$ nodes (within a single Bunyip group) in the fashion indicated in Figure 1.

Each timing result was averaged over 10 measurements; this ensured that there was a standard error of less than 3% in each sample.

The **%Mem** values were the maximum memory utilization reported by the Linux `top` utility. As noted earlier, there should be no significant page swapping occurring. The Linux `time` utility was used to report both the percent CPU utilization and the elapsed times for a single job. For j jobs, the `time` utility reported the elapsed time for all j jobs to complete, from which Equation 5 can be used to calculate the slowdowns.

p	N	Single job			Simultaneous jobs			
		%CPU	%Mem	Time (s)	2	4	6	8
LAM – local scheduling								
2	1500	72	5.2	6.8	1.85	3.50	5.30	6.91
4	2000	59	5.2	10.7	1.68	3.04	4.37	5.73
8	3000	40	5.8	24.6	1.62	2.65	3.66	4.49
16	4000	38	5.0	28.0	1.63	2.68	3.64	4.51
32	6000	26	5.8	61.8	1.58	2.41	3.24	3.95
SCore – gang scheduling								
2	1500	100	5.0	7.5	1.86	3.66	5.50	7.35
4	2000	100	5.2	11.3	1.91	3.79	5.71	7.61
8	3000	100	5.6	23.1	1.95	3.89	5.82	7.85
16	4000	100	5.1	26.9	1.95	3.88	5.85	7.85
32	6000	100	5.6	54.4	1.98	3.94	5.93	7.99

Table 2. $N \times N$ Linpack slowdowns, with p processes on p CPUs

4.3 Discussion

SCore achieved slowdowns of very close to j for j jobs. As there is some overhead for central co-ordination, one would expect these to be slightly greater than j ; that they are slightly less is likely to be due to the fact that processes will still yield for normal I/O. Even so, SCore’s gang scheduling seems to have very low overhead, and its slowdowns are as low as can be expected of a gang scheduling system.

For single job execution, SCore is somewhat faster for larger p , due to the fact that the process is busy-waiting, rather than yielding, when waiting for a message to be received (cf. Equations 4 and 1). This is particularly the case for Linpack, where, for the problem sizes selected, a significant amount of overhead is due to small messages.

However, for $p \geq 2$ and $j \geq 2$, local scheduling under LAM/Linux is faster in terms of absolute time, up to 80% faster for Linpack at $j = 8$ and $p = 32$.

For LAM/Linux, the matrix multiply slowdowns were only modestly less than SCore, and similarly were relatively insensitive to p . However, it is interesting to note that for $p = 2$ (where all communication is via the `sysv` transport), the slowdowns were less than one would expect considering that a single job is fully utilizing both CPUs (and thus must be busy-waiting while waiting for messages). This indicates that with multiple jobs, there must be some yielding occurring when waiting for messages, permitting some overlap between jobs.

However, the Linpack program showed reduced slowdowns as p (and hence the relative communication overheads) increased, especially for larger j . This seems to be only partially explainable by the fact that for $j = 1$, the CPU utilization is significantly less than for matrix multiply, Considering that Linpack is considerably more fine-grained, this is rather a surprising result.

For larger N , at $j = 8$, slowdowns abruptly increased as the total memory requirements of the jobs exceed available memory [17]. At this point, the effects of scheduling policy became irrelevant; the dramatic slowdowns observed earlier when several users’ parallel jobs competed on

the Bunyip were in fact due to this effect.

5 Performance Analysis of Time-Shared Parallel Jobs

In this section, we develop an optimistic performance model for time-sharing where a process waiting on a message is yielded, and compare LAM/Linux’s performance with that of the model. Using this, and a more detailed view of the experiments of Section 4, we can evaluate the assumptions of the model and the performance of LAM/Linux over the ideal. Finally, we will use the model to extrapolate the performance over a faster cluster than Bunyip.

5.1 An Optimistic Performance Model

With a parallel job’s execution time being given by $t_1(\alpha, \beta)$, where α and β are defined as in Section 2.1, the execution time for $j > 1$ simultaneous such jobs can be modelled by:

$$t_j = \max(t_1, jt_1(\alpha_c + \alpha_{CPU}, \beta_{CPU})) \quad (6)$$

where α_c is the cost of a context switch, and $\alpha_{CPU} + \beta_{CPU}n$ is the amount of process time spent in the CPU in the transmission of a message of length n . These can be determined by measuring the % CPU time spent in the ping-pong benchmark. Such experiments for LAM MPI using the Bunyip’s TCP/IP transport yielded:

$$\alpha_{CPU} = 28\mu s, \beta_{CPU} = .14\mu s \quad (7)$$

Note that for the single job case, the context switch overheads when waiting on a message will form part of the measured value of α ; it needs to be explicitly included in Equation 6, as it will not contribute to the measured value of α_{CPU} .

The model makes the following assumptions:

1. j is sufficiently large so that whenever a process waiting for a message is yielded, there is another process ready to run. This assumption implies a total CPU utilization of 100% over the j jobs.
2. performance degradation due to cache misses occurring on swapped jobs can be neglected.

5.2 Single Job Execution Time Models

Execution time of the (dominant) multiply computation of the matrix multiply program described in Section 4.1 is given by:

$$t_1 = \frac{2N^3}{PQ}\gamma_3 + \frac{N^2}{PQ}((P-1)\beta_v + (Q-1)\beta_h) \quad (8)$$

In the first term, γ_3 is the cost per (BLAS Level 3) floating point operation ($\frac{1}{\gamma_3}$ corresponds to 403 MFLOPs on the Bunyip).

The second term corresponds to the vertical and horizontal broadcast¹, via a ring-shift all-gather operation [14]. β_v and β_h are the vertical and horizontal communication costs per word taking into account the asymmetries when a $P \times Q$ logical grid is allocated on the Bunyip (cf. Figure 1). While in principle it would be possible to form expressions of these in terms of the β values for the TCP/IP and `sysv` transports (Equations 1 and 2), we use a benchmark program performing the same all-gather operation on an 8×4 grid to directly yield:

$$\beta_v = 1.73\mu s, \beta_h = 1.03\mu s \quad (9)$$

However, for $j > 1$ jobs, the effect of the asymmetries will be small if Assumption 1 holds, and both these values can be replaced by that of β_{CPU} .

In a similar way, the LU factorization computation with *storage blocking*, the dominant computation in the Linpack program, is modelled; the details have been published previously [13]. Noting that here, there should be somewhat less scope for contention in vertical communications than for matrix multiply (cf. Equation 9), we estimate the effect of the asymmetries to be:

$$\beta_v = \beta_h = 1.0\mu s \quad (10)$$

5.3 Comparison of Model to Experimental Results

Table 3 gives various slowdowns for the $p = 32$ results from Tables 1 and 2. From dividing the observed total % CPU utilization for 1 job by that across j jobs, we have a measure of slowdown which takes into account the actual degree of overlap of inter-job computation with communication, but not effects from cache pollution when process are switched. From this, we can estimate the validity of Assumption 2 (third row), it causes an error of the model within 10% for LU, and an almost negligible error for matrix multiply.

Comparing the experimental multiply and LU times with that of the models, we see a reasonably close agreement, indicating they are calibrated fairly accurately on the Bunyip.

Comparing these slowdowns gives the total error of the model (row 6); subtracting that of row 3 from this gives the error in the model due to Assumption 1 (row 7). This gives an upper bound on what improvement can be expected due to improving local scheduling policies. For LU, Linux's local scheduling comes quite close to this for $j = 8$. However, for matrix multiply, the degree of concurrency achieved is far lower than what seems possible in principle, and furthermore seems to level out at $j = 4$.

Why the matrix multiply program did not achieve the slowdowns that the Linpack program did is still unclear. Using smaller values of N , using smaller values of k and using an all-gather algorithm based on pipelined or tree broadcasts [14] only reduced the slowdowns marginally. Our conjecture is that multiply matrix multiply jobs will tend to synchronize at their communication stages; hence most times when a process is waiting on communication, the other processes will be similarly waiting, and CPU utilization will remain low.

For $j = 8$, the contribution due to the context switch component α_c of the model is effectively zero for both matrix multiply and LU.

¹For the simplicity of presentation, we have omitted the communication startup (α) term in Equation 8, as this makes $< 1\%$ contribution for the problem and grid sizes of interest.

	t_1 (s)	s_2	s_4	s_6	s_8
matrix multiply, $N = 4000$					
expt, job:	20.9	1.83	3.59	5.24	6.96
expt, %CPU:		1.86	3.49	5.15	6.87
% err, A. 2		0%	3%	2%	1%
expt, MM:	18.8	1.92	3.63	5.26	6.89
model, MM:	17.7	1.12	2.25	3.38	4.50
% err, total		42%	38%	36%	35%
% err, A. 1		42%	35%	34%	34%
Linpack, $N = 6000$					
expt, job:	61.8	1.58	2.41	3.24	3.95
expt, %CPU:		1.56	2.27	2.96	3.66
% err, A. 2		1%	6%	9%	7%
expt, LU:	58.6	1.56	2.35	3.09	3.83
model, LU:	56.2	1.00	1.69	2.54	3.38
% err, total		36%	28%	18%	12%
% err, A. 1		35%	22%	9%	5%

Table 3. Experimental and model execution time and slowdowns (and percentage errors between slowdown measures), for benchmarks under LAM/Linux on an 8×4 grid on the Bunyip cluster

The results of Table 3 were also generated for smaller data sizes ($N = 3000$ for matrix multiply and $N = 4000$ for LU). The $j = 1$ CPU utilizations were 5% and 15% respectively; this permitted smaller slowdowns – at $j = 8$, these were 6.44 and 3.69 respectively. Apart from this, all other trends were very similar to those in Table 3.

Apart from the fact that the Bunyip has a slow communication network, the models predict similar kinds of slowdowns for other kinds of clusters, for application having a similar proportion of communication to computation (indicted by single job CPU utilization) on the data and grid sizes of interest. For example, if we scale down communication volume coefficients by 10 and all other coefficients by 4 (roughly corresponding to a cluster with 2 GHz CPUs and 1 Gb network), the LU component of Linpack at $N = 6000$ on an 8×4 grid would have a 45% CPU utilization for a single job, with a predicted slowdown of 5.4 at $j = 8$.

6 Conclusions

Our studies have shown that on cluster computers, due to the fact that communication is treated as a form of I/O, local scheduling can significantly out-perform strict gang scheduling, in terms of overall throughput of multiple jobs, as the latter is not able to take advantage of the potential concurrency between simultaneous jobs. For this effect to occur, the application need not be coarse-grained, as in the case of Linpack; however, the presence of some large messages may be needed to gain a sufficient degree of concurrency.

This is not to say that gang scheduling is not a valuable component of cluster management systems such as SCore; it is still useful for efficient job packing (space-sharing). A useful extension of this in such systems would be to limit the active (i.e. ready-to-run) jobs based on total memory utilization [1]. Ultimately, by extending paired gang scheduling [18] to gangs of several jobs, high overall throughput could also be achieved.

However, a cluster management system that merely selected the best set of nodes for a new job (based on load and memory considerations) and relied on local scheduling would be almost as useful in practice, and much simpler to implement and deploy. Its main potential disadvantage is that it would not be able to avoid thrashing when the total memory requirements of submitted jobs exceed physical memory capacity.

From our analysis of local scheduling, the effect of (direct) context switch overheads was negligible; resulting overhead from cache pollution is likely to be more significant. Local scheduling's overall throughput advantages do not only apply for coarse-grained applications: in fact, in our studies, it performed much better on the finer-grained Linpack application, which had a mix of small and large messages. Indeed, taking cache pollution effects into account, near-ideal scheduling behaviour, as predicted by our simple performance model, was achieved by the Linux kernel.

Furthermore, the two applications studied showed that, when considering scheduling policies, both memory footprint and communication patterns have an important effect on overall throughput. These factors should not be neglected in future studies.

Future work would include more comprehensive evaluations, e.g. jobs of mixed running times, memory footprint and communication patterns. For a more complete understanding of local scheduling behaviour, the (Linux) kernel need some extension to improve infrastructure such as more accurate recording of CPU time for a process, implementing counts of events such as how often a process is yielded for I/O, and logging information on scheduling-related data. This may,

for example, give a clearer insight into why the matrix multiply slowdowns were much higher than expected. This would also form a framework for investigating how to optimize local scheduling strategies and techniques, such as setting optimal timeout values for wait-on-communication loops, and the immediate waking up of a process when message arrives.

Acknowledgements

The authors would like to thank Bob Edwards for providing system administration support, and Joseph Antony for installing SCore 5.4.0 on the Bunyip.

References

- [1] A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *14th International Parallel and Distributed Processing Symposium*, 2000.
- [2] The Bunyip (Beowulf) Project. <http://tux.anu.edu.au/Projects/Beowulf/>.
- [3] F. A. B. da Silva and I. D. Scherson. Improving parallel job scheduling using runtime measurements. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 18–38. Springer Verlag, 2000. Lect. Notes Comput. Sci. vol. 1911.
- [4] D. G. Feitelson. Scheduling parallel jobs on clusters. In R. Buyya, editor, *High Performance Cluster Computing, Vol 1: Architectures and Systems*, pages 519–533. Prentice-Hall, 1999.
- [5] D. G. Feitelson. Metrics for parallel job scheduling and their convergence. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.
- [6] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.
- [7] D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, 1996.
- [8] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of gang-scheduling on workstation cluster. In *JSSPP*, pages 126–139, 1996.
- [9] The Hebrew Univeristy - Parallel Systems Lab. the ParPar project. <http://www.cs.huji.ac.il/labs/parallel/>.
- [10] PC Cluster Consortium. SCore cluster system software. <http://www.pccluster.org/>.
- [11] U. Schwiegelshohn and R. Yahyapour. Fairness in parallel job scheduling. In *Journal of Scheduling*, volume 3(5), pages 297–320, 2000.
- [12] P. G. Sobalvarro and W. E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 106–126. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.
- [13] P. Strazdins. A Comparison of Lookahead and Algorithmic Blocking Techniques for Parallel Matrix Factorization. 4(1):26–35, Apr. 2001.
- [14] W. B. Tan and P. Strazdins. The Analysis and Optimization of Collective Communications on a Beowulf Cluster. In *The 2002 International Conference on Parallel and Distributed Systems*, pages 659–666, Taipei, Dec. 2002. IEEE Press.
- [15] The LAM MPI Team. LAM/MPI users guide. <http://www.lam-mpi.org/>, Sept. 2003.
- [16] A. Tridgell, P. Mackerras, D. Sitsky, and D. Walsh. AP/Linux - A modern OS for the AP1000+. In *Sixth Parallel Computing Workshop*, Kawasaki, Nov. 1996. Fujitsu Parallel Computing Research Center.

- [17] J. Uhlmann. Efficient Job Scheduling on Cluster Computers. Honours Thesis, Department of Computer Science, Australian National University, Nov. 2002.
- [18] Y. Wiseman and D. G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):591–592, June 2003.
- [19] B. B. Zhou, R. P. Brent, D. Walsh, and K. Suzaki. Job scheduling strategies for networks of workstations. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 143–157. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.