Gang Scheduling with Memory Considerations

Anat Batat Dror G. Feitelson

Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

A major problem with time slicing on parallel machines is memory pressure, as the resulting paging activity damages the synchronism among a job's processes. An alternative is to impose admission controls, and only admit jobs that fit into the available memory. Despite suffering from delayed execution, this leads to better overall performance by preventing the harmful effects of paging and thrashing.

1 Introduction

A major issue for parallel systems is how to divide the system resources among a number of competing jobs, giving a reasonable quality of service to all the system's users [11]. This allocation of resources is done by a central operating system which runs on a host that manages the whole system. The system scheduler determines when and on which nodes a job will be executed.

One way of scheduling parallel jobs is *gang scheduling* [7]. Its main idea is to map the threads of a parallel job to distinct processors, and then schedule them to be executed simultaneously on their respective processors. Thus, they either all run at once or none do. Using time slicing allows for interactive response times. The fact that all threads run simultaneously makes it possible for them to synchronize with each other [10]. The price of gang scheduling is that the overall system performance is not always optimal. There is some interference in the cache, and overhead for context switching. However, most studies find gang scheduling to be very efficient [13, 4, 14].

A major drawback of current implementations of gang scheduling is that they do not take memory requirements into account. On a regular computer (not part of a parallel system), it is possible to use more then the existing physical memory. The operating system handles this by paging and swapping. Today, when there is a tendency to run a whole Unix operating system on every node of a parallel system, paging can be had for free. However, paging is still undesirable, because the paging mechanism is not synchronized by nature [3]. Using it may harm the synchronization of the parallel program's threads. In order to avoid paging all the program's memory address space must be in the main memory.

The simplest way to prevent memory pressure is to run only a subset of the jobs, and delay the rest until the needed memory becomes available. The main question is whether the queueing delay will be bigger or smaller than the delays caused by the paging inefficiency. Our research indicates that the answer is that delaying processes execution is more efficient then running them all together, using paging.

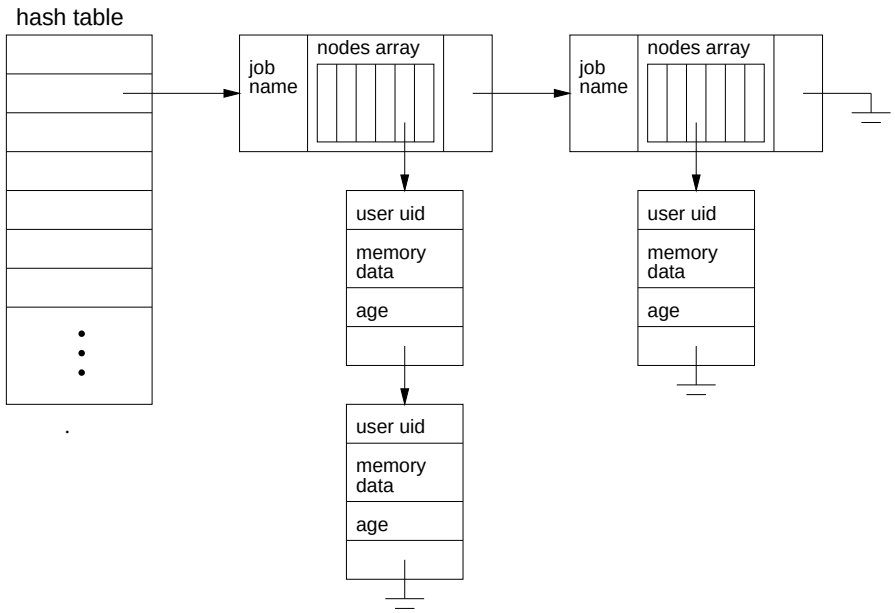

Figure 1: *Scheme of the data structure used to store and access the jobs' memory information.*

2 Memory Usage Estimation

To use addmission controls, one has to know in advance how much memory a job will use. In this section we are going to discuss two ways for estimating the memory space a job will consume. The first is estimation by using knowledge on previous executions of the job. If the job runs for the first time, we will have to estimate the memory in another way. In this case, we will use information on the static memory, which exists in the executable file.

2.1 Using previous information

2.1.1 Identifying the job

In order to use data on previous executions of a job, first we have to define how a job is identified; or in another words, we have to define when two jobs are identical enough in a way that there is a high probability they will consume the same amount of memory. Identity can be defined by the following three characteristics: file name (the executable path), number of nodes the job uses, and the name of the user [12]. If we don't have information on previous executions of a job according to this triple identification we can use information that exists for partial identification of the job: path + number of nodes, or even path only.

We want fast access (insert, find, and remove) to fully and partially identified jobs, so we use a hash table. The data structure is described in Figure 1. We need to hash every job by three keys or less. Therefore we maintain a hierarchy of three pointers. When a new job arrives, we know its name, the number of nodes it uses, and the user identification. We would like to access the data of

the jobs with these keys. The hash table hashes the executable name to a struct. This struct's fields include a pointer to the slot of the next job that will be hashed to the same hash entry. Another field is an array, such that its *i* cell represents jobs with the same name that used *i* nodes. This cell points to a linked list of other structs of all the users that run this file with *i* nodes. The struct's fields include data on the job's memory and the age of the job (the age of a job indicates how many days passed since the entry was last used). By scanning the user list and summing the memory usage we can get data according to a partial key of path + number of nodes. We can also access all the jobs with a specific name, by scanning all the users lists linked to entries of the nodes number table. In practice, the data structure is implemented by 3 tables, to avoid dynamic memory allocation. This data structure's total memory size (measured according to its implementation on the ParPar system) is 320K, where the maximum number of different job's paths is 2048, and the maximum number of different jobs (path + user) is 4096.

2.1.2 Data collection

The data collection is done on the nodes. The memory consumption of every process is then sent to the host. In principle, the memory usage of each process can be found from the `rusage` struct returned by `wait4` after it terminates. Unfortunately, these fields are not filled in all the version of Unix that we checked. Therefore we resort to reading the kernel data structures during the process's execution, as is done by the `top` utility. The measurements are done once a second. As memory usage is non-decreasing, only the last value is stored. When the process ends its run, the stored value is sent to the host with other information about the process and its run. Special actions must be taken to handle short processes. A short process execution time might end before the first measurement after its execution. Therefore another measurement is done right after an execution of a process (after the call to the `exec` function).

In order not to clog the system with stale data, it is necessary to get rid of old information. To do so, each jobs' memory information will be divided into two: current month information and previous month information. At the end of a month, the current month data will become the previous month data and the collection of the new month's information will be started. In case the job hasn't been run for two months, its entry is removed. This way, the information that is used to estimate a job's memory usage is not older than two months and at least the last month executions' information, if such exist, is available. This algorithm can be implemented easily, by keeping two memory-data fields in the user's struct, and once a month scaning all the users' lists to update the memory data fields.

2.1.3 Estimation function

The value we produce using estimations is most likely to be inaccurate. Estimating a value too high or too low, compared to the correct one, has disadvantages. A too high value may cause a postponement of the execution of our job or later arriving jobs due to the misconception that the available memory space is insufficient. The system resources utilization will be damaged as a result of such estimation. On the other hand, a too low value is likely to cause excessive paging

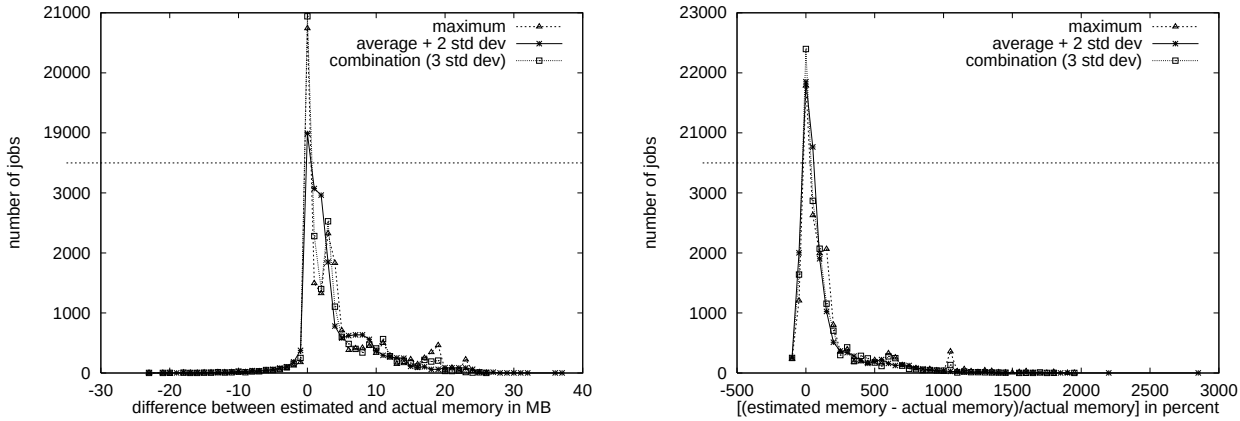

Figure 2: *Comparison between the performance of the different estimation functions (note the split linear scale).*

and swapping, either because the job will consume more memory than the nodes can supply, or because new jobs will be scheduled according to an over evaluation of the free memory space, which (in both cases) will slow down the node and along with it, the whole system.

To estimate the memory usage of a job, we need a statistical function that will process previous data to a single value. In choosing the function we should keep in mind that we prefer to over estimate memory usage and not the opposite. We chose to focus on three functions:

1. The maximum of all previous values
2. The average of previous values + 2 standard deviations (avg+).
3. A combination of the two: the minimum between the maximum value and the average + 3 standard deviations.

To check the quality of the different estimation functions, we checked their performance using a log file from Los Alamos National Lab. It contains 9 months (January through September of 1996) documentation of executions of approximately 50,000 jobs on a Connection Machine CM-5 parallel system. The collected data includes executable file name, running time, user name, number of nodes, and memory size. We predicted the memory size of every job according to previous similar executions (file + number of nodes + user name) or partial ones (file + nodes number, or file only), if there were any.

The results are shown in Figure 2. We divided the processes into buckets, according to the difference between the predicted value and the real value. The first graph is a histogram of the number of processes in every bucket: the y axis shows the number of processes in each bucket, and the x axis shows the difference between the predicted values and the real values. Each bucket has a range of 1MB. The second graph shows the relation between the predicted and real values: the x axis shows the differences between predicted and the real memory size as a percentage of the real size, when the range of each bucket is ten percentage points.

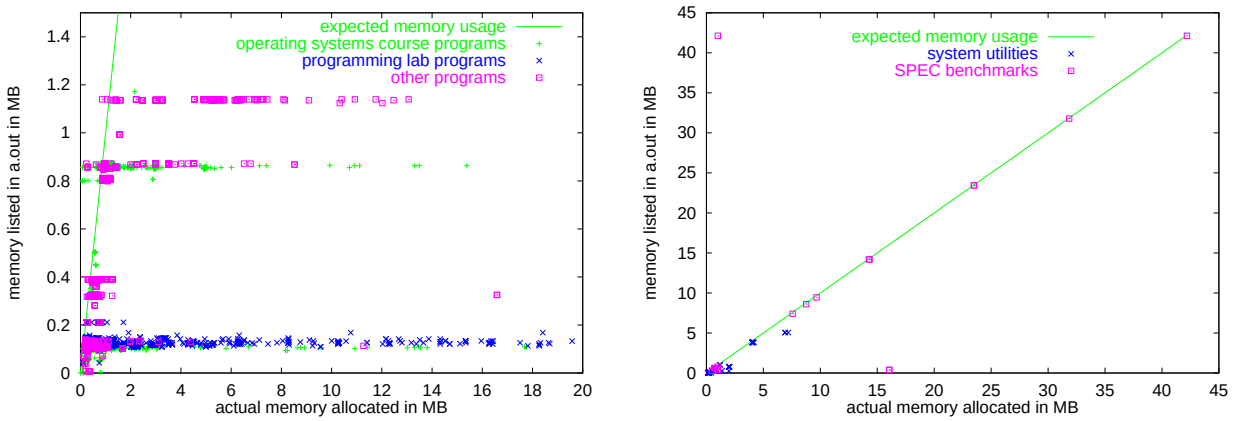

Figure 3: *Static memory against actual memory of students' programs (left; the scale actually extends up to about 40MB) and of system utilities and SPEC benchmarks (right).*

Observing the graphs, we can see that a high number of processes centralized near 0 indicating very good predictions. In the combination graph in about 87% of the processes the absolute difference is smaller then 5MB (compared to 85% for the other functions). In 70% of the processes the absolute difference is smaller then 1MB (compared to 66%). We can also see there are less processes on the negative part of the graph, where the estimated value is lower than the real value (in the combination graph 6% of the processes are in that range, in the maximum graph 4%, and in the avg+ graph 7%).

2.2 a.out memory information

We don't always have information from previous runs. The alternative is to find static information from the executable. Every executable file in `a.out` [1] format contains in its beginning a struct with compiler-known information on the executable. This includes information about the sizes of the text segment, the data segment, and the bss.

To check the accuracy of `a.out` data as a predictor of the program memory usage, we measured the actual memory usage of processes and compared it with their `a.out` information. We measured three different classes of processes.

- *Students' programs*

  We checked programs of the Computer Science students which were run on the Hebrew University computers. The measurement results are shown in Figure 3 (left). As can be seen in the graph, the static memory size depended on the student's course rather then on the actual memory size. Most of the programs were written in C++, and therefore included lots of dynamic allocations. This can explain the poorness of the results.

- *System utilities*

We checked programs of system utilities that were run by users or by the operating system (applications like: netscape, csh, latex etc.). Most of these programs are written in C. As can be seen in Figure 3 right, the correlation between the programs' static memory and the actual memory is better then for the students' programs.

- *Scientific programs*

  As representatives of this group we checked the SPEC benchmark programs. The result are also shown in Figure 3 right. As can be seen on the graph, there is a strong connection between the static memory and the actual memory. In most of the programs the two sizes are very close. However, it should be remembered that the results from the previous section indicate that in real parallel systems the situation is not so rosy — if jobs always used the exact amount of memory indicated in their executable, the predictions would always be perfect.

3 Scheduler Implementation

The idea of queueing jobs that do not have enough memory available was implemented in the framework of the ParPar system [7]. This section first describes the system, and then the implementation of the memory considerations.

3.1 The ParPar system

3.1.1 Hardware

The ParPar prototype cluster is built from 17 PCs: A host called "parpar" and 16 nodes named from "par1" to "par16" (Figure 4). The PCs have an Intel Providence motherboard with a Pentium Pro 200 processor, 64 MB DRAM, and a 2.1 GB SCSI disk. The nodes are connected by two independent networks. One is a switched Ethernet which serves as a control network. All the system oriented communication passes through it, using the conventional TCP/IP protocol and a reliable multi-cast protocol developed locally [5]. The other network is a 1.28 GB/S Myrinet dedicated to the users' applications communication (this network is called the data network). The communication is done using MPI over FM [17]. The data network connects only the 16 nodes (user processes don't run on the host).

3.1.2 Software structure

The ParPar software is based on the Unix BSDI system, and runs at user level. The software components of the ParPar system includes daemons which run on the host and on the nodes, and graphical user interfaces which run on the user's workstations (Figure 4). In addition there are the user processes which make up the parallel applications which are executed by the system. Single processes are called *sprocs*, the group of processes which composes a job is called an *xproc*.

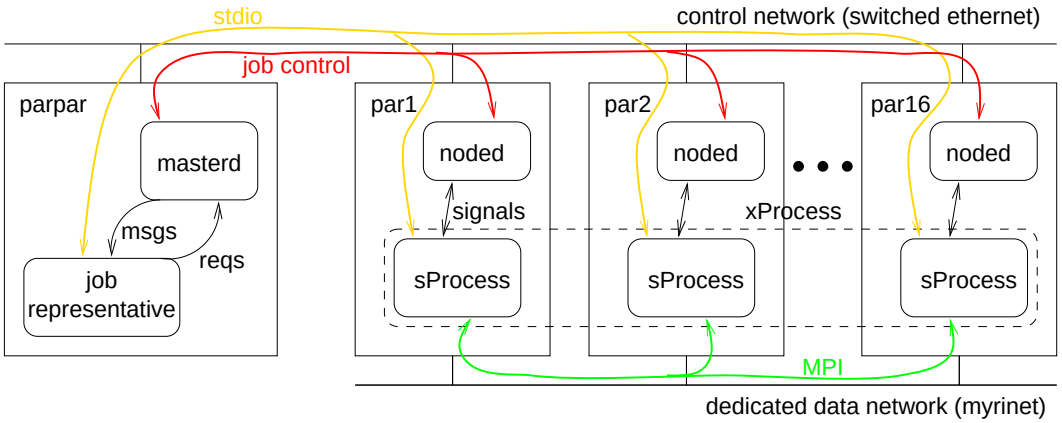

Figure 4: *Hardware and software components of the ParPar system.*

The master daemon (masterd) is the heart of the system. It is responsible for tracking the configuration of the system (which nodes are up and part of the system), for job management, and for resources allocation. In particular, the job scheduling we are talking about is done by the masterd. There is only one masterd in the system and it runs on the host.

The node daemons (noded) provide local control over the running processes on the system's nodes. As such they are the masterd's agents. There is one noded on each node.

The job representative (JR) is a GUI which provides an interface to the system and terminal I/O facilities. The users use the JR to run jobs by naming the executable and the arguments, and providing a list of optional partition sizes to use. The system displays status messages on the JR. Normally there are different JRs for every job running on the system. The JR can run on the host or on the user's workstation. A textual JR also exists, to run batch scripts.

3.1.3 Gang scheduling

The ParPar system uses gang scheduling as it's scheduling scheme. Gang scheduling combines time slicing with space slicing. The main idea is that the processes of a parallel job are mapped to distinct processors, and are then scheduled to execute simultaneously on their respective processors. Thus they either all run at once, or none do.

In order to map the processes to processors, we use Ousterhout's matrix algorithm, where columns represent nodes and rows are scheduling slots [16]. Packing of jobs into slots is done using a buddy system, as was proposed in the context of the Distributed Hierarchical Control scheme [8, 9]. This tends to allocate processors in pre-defined groups that are powers of 2, also leaving such groups free for future use.

After we estimated the memory size of a job, we need to add this information to our scheduling decisions. We need a function that will take into consideration all factors — previous factors which are built into the current scheduling algorithm, and the new memory factor. The original gang scheduling algorithm considers only the load factor — the number of running processes on a node. It selects the least loaded controller such that a time slot exist in which all the controller's

nodes are free. To add memory load considerations, we also check that each of these nodes satisfies

$$process\ memory \leq (the\ node's\ physical\ memory) * C - occupied\ memory$$

Where C is a constant bigger then 0. C limits our maximum used memory. It is not trivial that the case when $C = 1$ gives the best results. The best size is clearly around 1, but a slightly higher or lower size might work better then the exact 100% of free memory. One of our research goals is to find the best size.

If there are not enough nodes with sufficient memory, the job will be pushed into a queue until the required memory will be available, or until the system total load will be zero (to make it possible to run processes which consume more then the initial total memory).

When a job finishes its execution the queue is scanned for the first job that can be executed, i.e. there are enough nodeds with enough available memory (more or equal to its estimated memory usage) to execute all its processes. This procedure is repeated until there is not enough memory to run any additional jobs. To avoid starvation a job in the queue cannot be skipped more then 15 times. If a job was skipped 15 times no job will be executed until this job will.

The queue and its management already existed in the ParPar system. The memory considerations full implementation was added to the ParPar resource management module, in the function that chooses the controller and slot.

4 Experimental Results

In order to check the efficiency of our ideas, we compared the system performance with and without our improvements. We checked the effect of memory pressure on performance, comparing the influence of different degrees of memory pressure. In cases of lack of available memory space, a major factor that influences the system performance is the locality of the program's memory accesses. Therefore we ran programs with several locality levels.

4.1 Workload

Our workload was composed of synthetic applications that were executed on the ParPar system. This is not a simulation but a real execution of multiprocess jobs. Thus all the paging effects are real and measured directly, and we do not need to model them.

We chose to use synthetic programs instead of existing benchmarks for several reasons. The main reason is a desire to check very specific issues. Real programs behavior is influenced by lots of factors. Using synthetic programs lets us restrict the influence of irrelevant factors and limit the checking to issues that interest us. Another reason not to use benchmarks is that there are no agreed benchmarks, especially for memory usage.

The programs we created use lots of memory, enough to cause memory pressure by executing only a small number of jobs. The programs also have a varying degree of locality. The locality degree and the program's run time are determined by arguments the program gets.

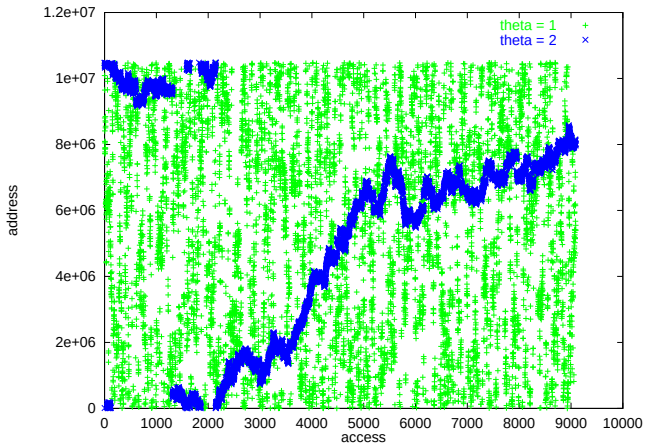

Figure 5: *The spread of 90909 random accesses with low and high degrees of locality.*

To construct programs with different degrees of locality, we used the fractal model [18]. According to this model, the dynamic behavior of a program can be modeled as a one-dimensional fractal random walk. The model is based on the parameterization of a program behavior by a small number of constants, one of which is the fractal dimension of the program. A program's memory accesses can be simulated by generating random numbers $X(t)$, where $X(t)$ indicates the jump from the address accessed at time t to the address at time $t + 1$. The distribution of $X(t)$ is given by:

$$Pr[X(t) > u] = Bu^{-\theta}$$

Where t is the virtual time marked by the ticks of the accesses to memory, B is a normalizing constant (we chose $B = 1$) and θ is the locality degree of the program's memory accesses and satisfies $1 \leq \theta \leq 2$. θ = 1 produces the lowest locality and θ = 2 produces the highest locality. Figure 5 shows the spread of 90,909 random accesses, when θ = 1 and when θ = 2.

Figure 6 shows the main part of the code of the program. The program uses a static array of 10MB. This size equals K (=1024) times the constant RANGE, where RANGE is the range of addresses produced by 100,000 accesses with θ = 2. Choosing a bigger size will prevent full use of the array with a high degree of locality (θ = 2), unless the number of accesses is increased. The program produces random jump sizes and accesses an array cell on the K multiple of the new address (previous address + jump). The size of the jump depends on θ. When θ = 2 (the highest locality) 94% of the jumps' sizes are less than 4. Two successive memory accesses with such a jump will be in the range of 4K, thus in a range of a page. When θ = 1 (the lowest locality), only 75% of the jumps' sizes are less than 4.

4.2 Memory load and locality

The first experiments used a static workload on a reduced version of the ParPar: masterd and two nodeds.

```
char mem[K*RANGE];

for (i=0 ; i<max ; i++) {
   randint = random();
   r = ((double)randint) / maxrand;
   jump = pow( r, theta );
   if (randint & 0x01)
      jump = -jump;
   addr += jump;
   intaddr = (int)addr;
   intaddr = intaddr % RANGE;
   if (intaddr < 0)
      intaddr += RANGE;
   intaddr *= K;
   mem[intaddr] = intaddr % 200;
}
```

Figure 6: The code of the program that was used in the experiments.

The first experiment was run once without limiting the memory pressure, and again when limitating the memory load to 100% and 150%. We considered all the non-kernel memory as our free memory. This size might be exaggerated. Although the nodeds run on dedicated processors, some other processes (system processes) also run there. These processes memory usage can reach up to 20% of the non-kernel memory. The size of the main memory of a node machine is 64MB. About 55MB of it is the non-kernel memory, and about 45MB of those 55MB are actually free.

We measured the execution time of different numbers of jobs from 5 jobs till 50. Every job runs on two nodes. Every instance of the job used 10MB of memory space on each node. Thus, the available memory on a node can contain 4 jobs at most. The experiment was repeated several times with different degrees of locality of memory accesses. Figure 7 shows 6 different graphs, one for each combination of memory pressure and locality level. The graphs show execution time as a function of number of jobs (= memory load).

From Figure 7 it can be seen that when the memory pressure is limited by our mechanism the graph is linear. When there is no memory limit, the graph's gradient grows super-linearly with the number of jobs. It also can be seen that the locality degree has a big influence on the results when the system uses more memory than available. The most important result is that for a given θ the run time of the jobs is consistently smaller when the improved version of the system is used, that is, when the maximum used memory is limited and the jobs are put in a queue until enough memory will be available (in contrast to the case when the used memory is not limited and the UNIX operating system has to deal by itself with the memory overload).

In order to check what is the effect of memory pressure on performance, we ran the ParPar system with different limitations of the maximum memory pressure, from 80% till 200%. As has

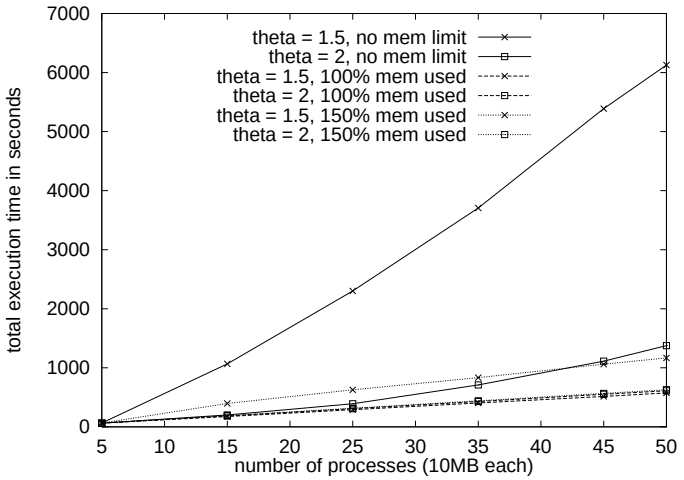

Figure 7: *Execution time as a function of number of processes, for different combinations of memory pressure and locality level.*

been mentioned, the size we refer to is the initial free memory which might be 20% higher then the actual size. Therefore a maximum memory pressure of 80% was also measured. We measured the execution time of 50 jobs with the same locality level. Each job runs on two nodes, and every instance of the job uses 10MB of memory space. The experiment was repeated with different locality levels. Every noded has about 55MB available memory, thus, the jobs needed about 900% of its memory. Figure 8 shows different graphs for the different memory pressures. Each graph shows execution time as a function of locality (theta).

From Figure 8 it can be seen that low locality causes long execution time. It is also obvious

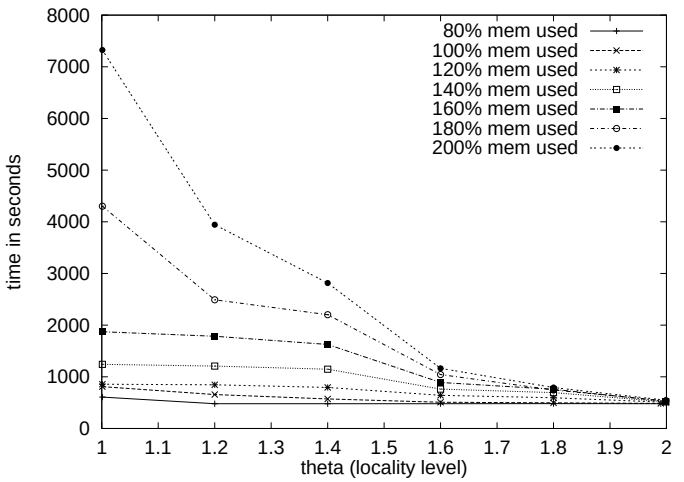

Figure 8: *Execution time as a function of locality level, for different memory pressures.*

that when the locality level is low, the differences between the execution run time of processes that run on systems with different memory pressures are more significant than in the same situation, but with higher locality. Processes with the lowest locality suffered a deceleration of 800% when the maximum allowed memory load was 200% compared to the situation with a maximum load of 100%. On the other hand, the run time of processes with high locality and different memory load was almost identical.

4.3 Paging vs. queueing

In the experiments described above, all the jobs are run together and every one of them uses all the nodeds in the system. Thus, the system is fully used all the time. Under these conditions, the influence of memory management on job scheduling is not checked. A possible delay in the execution of jobs is not brought into consideration. Such a delay can damage the scheduling and decrease the system performance.

We therefore also tried to check the efficiency of our ideas under more realistic conditions — on a system where the arrival time, the number of nodes, and the run time of the jobs change. We used the full size system: masterd + 16 nodeds. We ran 1000 jobs of 10MB each. The jobs had different run times, different numbers of nodes, and different arrival times. The distributions of these parameters were based on the workload model developed by Uri Lublin [15]. We checked different locality degrees of memory accesses. The experiment was repeated for different loads, once when the memory load was not limited and once with limitation of memory load to 100% of the real memory. The system load is defined as:

$$load = \frac{average\,run\,time * average\,number\,of\,nodes}{average\,inter\,arrival\,time * number\,of\,nodes\,in\,the\,system}$$

The results are presented in figure 9. The graphs present average response time and average slowdown as a function of system load. Response time is the total time the job spent in the system, from its arrival time till its execution is completed (including the time it spent in the queue). Slowdown is the relation between actual run time (response time) and expected run time.

It can be seen that limiting memory load consistently decreases the response time. The slowdown on the other hand, is highest when the maximum memory load is limited. The reason for these high values is a high slowdown of short jobs which are forced to wait in the queue. The percentage of queued jobs starts to grow when the load becomes more than 0.5, and reaches 80% when the load is 0.8.

Another interesting and important result is the existence of a critical load in which there is a sharp raise in the average response time, when the system was run without limiting memory load. Under higher loads the system can not handle the job stream. This critical point exists for every locality level. As the locality of the memory accesses is higher, it happens at higher loads. Above this point, the system is saturated. Figure 10 presents the number of processes currently running as a function of time, when the offered load is 0.7 and θ is 1.5. When the used memory is not limited the number of processes currently running increases until the last process arrives. The number of processes stays high for long time until the processes manage to finish their execution.

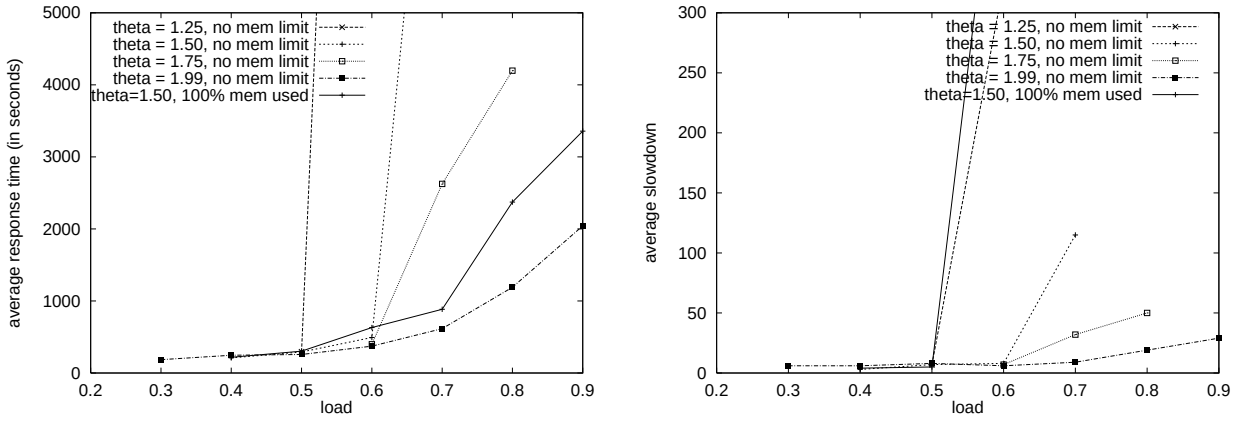

Figure 9: *Average response time and average slowdown as a function of load, for different locality degrees.*

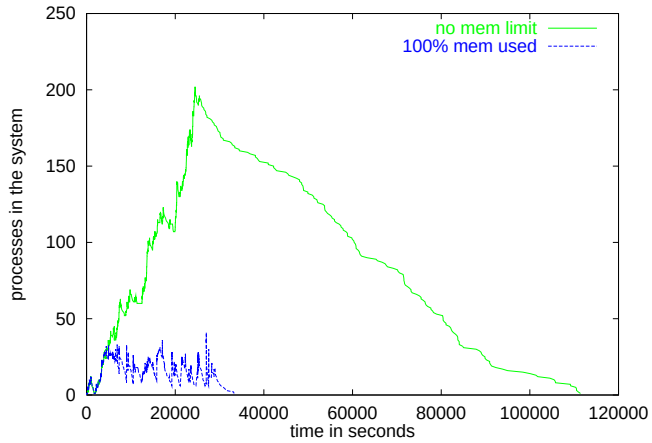

Figure 10: *Number of processes currently in the system as a function of time*

This means that the system was actually saturated, and the measurement succeeded only due to the finite number of jobs used. Unlike it, when the memory load is limited to 100%, the number of running processes was much more stable and it does not depend on the arrival time of the last process.

Finally, we wanted to verify the assumption that using more memory then available is more harmful then not using all the available memory. To check this we ran a partial version of the system (8 nodeds + masterd) limiting the used memory to 60%, 80%, 100% and 120% of the non-kernel memory. We used a fixed θ of 1.5. The system available memory is between 80% and 100% of the non-kernel memory, depending on the the memory usage of the system background processes. We assume that using a mistaken size of the available memory is equivalent to a similar mean error in the estimation of the memory usage of the processes. Thus, using 80% of the available memory emulates a case when the average size of the estimated memory usage is 120%

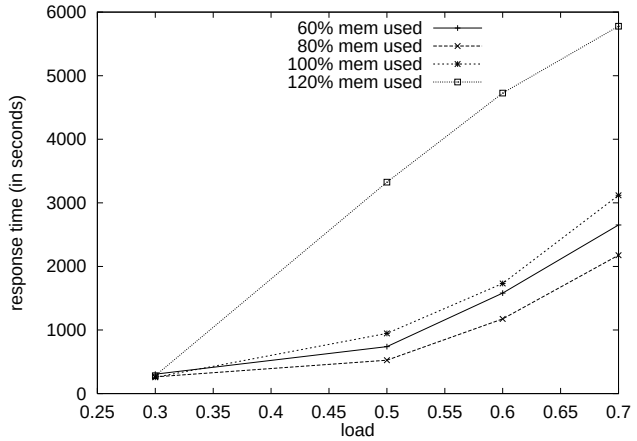

Figure 11: *Average response time as a function of load, for different memory pressures.*

of the actual usage.

Figure 11 presents the experiment's results. The graphs shows the response time as a function of load, for different memory limitations. It can be seen that the results for a limit of 80% are better then the results for 100%. As was explained before, 80% of the non-kernel memory is about 100% of the available memory, when the system processes are brought into consideration. Therefore the 100% of the non-kernel memory is 120% of the available memory and 60% of the non-kernel memory is 75% of the available memory.

It can be seen that when the system available memory is under estimated (60%), the system performance is better than when the available memory is over estimated (100% or 120%). These results verify our assumption that the delay caused by a partial use of the memory, is lower then the delay caused by using more than the available memory.

5 Conclusions and Future Work

Thrashing by nature damages the performance of a system, but for a system that runs parallel jobs the damage is much worse. The main question we faced was how to add memory considerations to the system's scheduler, which would prevent thrashing and also would not harm the system's utilization.

Our solution was to estimate the memory usage of newly arrived jobs, using information on their previous runs or on their static memory consumption. A job is executed only on a node that has enough available memory for its estimated memory usage. If there are not enough nodes that fulfill this demand for all the job's processes, the job is inserted into a queue until the needed memory will be vacated.

Our ideas was fully implemented and added to our parallel system (the ParPar system). We ran tests to check the efficiency of our ideas. The results showed that the system performance improved significantly after we added the implementation of our ideas. When the maximum memory load

of the system was not limited the system couldn't stand high loads and thrashed. Limiting the maximum memory load prevented the thrashing and the system managed to handle higher loads. The best performance was gained when the maximum used memory was exactly the available physical memory. When the used memory was limited the system performance was consistently better then the performance when there was no limitation, including in low loads, in which the system did not thrash.

These results show that our technique of adding memory considerations is efficient and can significantly improve the system performance.

Still, there is further work to be done. The biggest disadvantage of our proposed method is not specifically handling limitation of the time short jobs wait in the queue. The slowdown of a job depends on the relation between its pure run time and the whole time it spends in the system, i.e. its actual run time + the time it spends in the queue. Therefore, the time it spends in the queue has a big influence on its slowdown. There are two possible solutions to limit the queue wait time:

1. To use swapping. If a job runs too long time we can swap it out and load other job, which might be shorter [2].
2. To add run time considerations to the scheduler similarly to the memory considerations [12], and give priority to short jobs.

Another improvement to our work can be handling very big jobs whose memory usage was underestimated. Such processes can thrash the system. These jobs should be detected, swapped out of memory and their memory usage should be re-estimated.

Acknowledgements

Many thanks to the System Group, and especially Tomer Klainer, for their technical help in carrying out this research. Thanks also to Curt Canada of LANL for the CM-5 log, which is now available on-line [6]. This work was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences & Humanities, and by the Ministry of Science.

References

[1] BSD man pages.

[2] G. Alverson, S. Kahan, R. Korry, C. McCann, and B. Smith, "*Scheduling on the Tera MTA*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 19–44, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[3] D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood, "*Paging tradeoffs in distributed-shared-memory multiprocessors*". *J. Supercomput.* **10(1)**, pp. 87–104, 1996.

[4] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, "*Multiprogramming on multiprocessors*". In 3rd *IEEE Symp. Parallel & Distributed Processing*, pp. 590–597, 1991.

[5] D. Er-El, A. Kavas, and D. G. Feitelson, *Design, Implementation, and Impact of Multicast in the ParPar Control Network*. Technical Report 98-14, Hebrew University, Dec 1998.

[6] D. G. Feitelson, "*Parallel workloads archive*". URL http://www.cs.huji.ac.il/labs/parallel/workload/.

[7] D. G. Feitelson, A. Batat, G. Benhanokh, D. Er-El, Y. Etsion, A. Kavas, T. Klainer, U. Lublin, and M. A. Volovic, "*The ParPar system: a software MPP*". In *High Performance Cluster Computing, Vol. 1: Architectures and Systems*, R. Buyya (ed.), pp. 754–770, Prentice-Hall, 1999.

[8] D. G. Feitelson and L. Rudolph, "*Distributed hierarchical control for parallel processing*". *Computer* **23(5)**, pp. 65–77, May 1990.

[9] D. G. Feitelson and L. Rudolph, "*Evaluation of design choices for gang scheduling using distributed hierarchical control*". *J. Parallel & Distributed Comput.* **35(1)**, pp. 18–34, May 1996.

[10] D. G. Feitelson and L. Rudolph, "*Gang scheduling performance benefits for fine-grain synchronization*". *J. Parallel & Distributed Comput.* **16(4)**, pp. 306–318, Dec 1992.

[11] D. G. Feitelson and L. Rudolph, "*Parallel job scheduling: issues and approaches*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–18, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[12] R. Gibbons, "*A historical application profiler for use by parallel schedulers*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 58–77, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[13] A. Gupta, A. Tucker, and S. Urushibara, "*The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 120–132, May 1991.

[14] S. T. Leutenegger and M. K. Vernon, "*The performance of multiprogrammed multiprocessor scheduling policies*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 226–236, May 1990.

[15] U. Lublin, *A Workload Model for Parallel Computer Systems*. Master's thesis, Hebrew University, 1999. (In Hebrew).

[16] J. K. Ousterhout, "*Scheduling techniques for concurrent systems*". In 3rd *Intl. Conf. Distributed Comput. Syst.*, pp. 22–30, Oct 1982.

[17] S. Pakin, V. Karamcheti, and A. A. Chien, "*Fast messages: efficient, portable communication for workstation clusters and MPPs*". *IEEE Concurrency* **5(2)**, pp. 60–73, Apr-Jun 1997.

[18] D. Thiébaut, "*On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio*". *IEEE Trans. Comput.* **38(7)**, pp. 1012–1026, Jul 1989.