

Dynamic Load Balancing in MPI Jobs

Gladys Utrera, Julita Corbalán, Jesús Labarta

Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)
{gutrrera, juli, jesus}@ac.upc.es

Abstract. There are at least three dimensions of overhead to be considered by any parallel job scheduling algorithm: load balancing, synchronization, and communication overhead. In this work we first study several heuristics to choose the next to run from a global processes queue. After that we present a mechanism to decide at runtime whether to apply Local process queue per processor or Global processes queue per job, depending on the load balancing degree of the job, without any previous knowledge of it.

1 Introduction

Scheduling schemes for multiprogrammed parallel systems can be viewed in two levels. In the first level processors are allocated to a job, in the second, processes from the job are scheduled using this pool of processors. When having more processes than processors allocated to a job, processors must be shared among them.

We will work on message-passing parallel applications, using the MPI [16] library, running on shared-memory multiprocessors (SMMs). This library is worldwide used, even on SMMs, due to its performance portability on other platforms, compared to other programming models such as threads and OpenMP and because it may also fully exploit the underlying SMM architecture without careful consideration of data placement and synchronization. So the question arises whether to map processes to processors and then use a local queue on each one, or to have all the processors share a single global queue. Without shared memory a global queue would be difficult to implement efficiently [8].

When scheduling jobs there are three types of overhead that must be taken into account: load balance, to keep the resources busy, most of the time; synchronization overhead, when scheduling message-passing parallel jobs; and communication overhead, when migrating processes among processors losing locality.

Previous work [24] has studied the synchronization overhead generated when scheduling processes from a job need to synchronize each other frequently. It proposes a scheme based on the combination of the best benefits from Static Space Sharing and Co-Scheduling, the Self-Coscheduling. The communication overhead is also studied in [25] when applying malleability to MPI jobs. In this work we present a mechanism, the Load Balancing Detector (LDB), to classify applications depending on their balance degree, to decide at runtime the appropriate process queue type to apply to each job, without any previous knowledge of it.

We evaluate first several heuristics to decide the next process to run from the global queue depending on the number of unconsumed messages, the process timestamp, the sender process of the recently blocked or the process executed before the recently blocked one if it is ready. We obtained the best performance when selecting the sender process of the recently blocked process or if is not possible the one with the greater number of unconsumed messages.

When a process from a parallel job arrives to a synchronization point and cannot continue execution, we do blocking immediately and then context switching to a process in the ready queue. We present a novelty ratio to estimate load balance from a job by observing the coefficients of variation of the number of context switches and the user execution times, among processes. We constructed an algorithm that calculates these at runtime, and after deciding the type of application (balanced or imbalanced), applies immediately the appropriate type of queue. Results show that we obtained better performance when choosing the appropriate queue type at runtime, than applying a predefined queue type to all the applications without taking into account their balance degree.

The rest of the paper is organized as follows. In Section 2 we discuss the related work. Then in Section 3 the execution framework and in Section 4 follows the scheduling strategies evaluated. After that in Section 5 we show the performance results. Finally in Section 6 the conclusions and the future work.

2 Related work

There is a lot of work in this area; here we mention some work related to implementations of local and global queues from the literature.

Using local queues of processes at each processor is a natural approach for distributed memory machines. It is also suitable to shared memory machines as there is some local memory as well. Provided that only one processor will use each local queue, there won't be no contention and no need to locks. However there must be a decision where to map processes to processors and how to schedule them to satisfy the communication requirements to minimize the synchronization overhead.

There is an interesting and extensive discussion about using local and global queues in [8]. Local queues have been used in Chrysalis [14] and Psyche [19] on the BBN Butterfly. The issue involved in the use of local queues is load balancing, by means of migrating processes after they have started execution. The importance of load balancing depends on the degree of imbalance of the job and hence of the mapping. Then a scheduling algorithm must ensure the synchronization among processes [24][2][18][3][17][9]. There is also an interesting proposal in [5] where they do load balancing by creating threads at loop level.

A global queue is easy to implement in shared memory machines. It is not the case for distributed systems. The main advantage of using such queues is that they provided automatic load balancing or named load sharing as in [8]. However this approach suffers from queue contention [1], lack of memory locality and possible locks overhead, which is eliminated local per-processor, thereby reducing synchronization overhead. Other work in process scheduling has considered overhead

associated with reloading the cache on each context switch when a multiprocessor is multiprogrammed. [21] argued that if a process suspends execution for any reason, it should be resumed on the same processor. They showed that ignoring affinity can result in significant performance degradation. This is discussed also in [26].

Global queues are implemented in [6]. Here they implement a priority queue based on the recent CPU usage plus a base priority that reflects the system load. However this issue is not crucial as in most cases the cache is emptied after a number of other applications have been scheduled [12].

A combined approach is implemented in [7], where at each context switch a processor would choose the next thread to run from the local queue or the global queue depending on their priority.

About job classification at runtime there is a work in [10], where they classify each process from a job and varying the working set when applying gang scheduling.

3 Execution Framework: Resource Manager (CPUM)

Here we present the characteristics of our resource manager that is the Cpu Manager (CPUM). It implements the whole mechanism to classify applications depending on their balance degree and takes all the necessary actions to apply the appropriate queue type to each job. It also is in charge of deciding the partition size, the number of folding times depending on the maximum multiprogramming level (MPL) established, the mapping from processes to processors.

The CPUM is a user-level scheduler developed from a preliminary version described in [15]. The communication between the CPUM and the jobs is done through shared memory by control structures.

In order to get control of MPI jobs we use a dynamic interposition mechanism, the DiTools Library [20]. This library allows us to intercept functions like the MPI calls or a system call routine which force the context switch among process and is invoked by the MPI library when it is performing a blocking function.

All the techniques were implemented without modifying the native MPI library and without recompilation of the applications.

The CPUM wakes up periodically at each quantum expiration and examines if new jobs have arrived to the system or have finished execution, updates the control structures and if necessary depending on the scheduling policy, redistributes processors, context switch processes by blocking and unblocking them or change based on the mechanism proposed in this article the process queue type to each job.

4 Processor sharing techniques evaluated

In this section we describe the main characteristics of each scheme used for the evaluation as well as the LDB.

4.1 Local queues

As the number of processors assigned to a job could be less than its number of processes, there may be a process local queue at each processor. We choose the next process to run in a round robin fashion, and as soon as a process reaches a synchronization point where it cannot proceed execution (e.g. an MPI blocking operation), it blocks immediately freeing the resource.

Notice that processes in local queues are assigned to the processor during the whole execution, avoiding migrations and preserving locality.

4.2 Global queues

While in Local queues we have a process queue assigned fixed to each processor, in Global queues we have a unique queue for each application assigned fixed to a processor partition. The unassigned processes are kept in the global queue until the scheduler select them to run on a newly freed processor.

We implemented several heuristics to choose the next process to run, which we describe below:

- *Timestamp*: Each process has a timestamp which is incremented when it is waiting in the global queue, in order to do aging. The selected process will be the one with the greater timestamp. This turn out to be the classical round robin.
- *Unconsumed messages*: The selected process will be the one with the greater number of unconsumed messages.
- *Sender*: The selected process will be the sender of the recently blocked process if it has also the number of *unconsumed messages* equal or greater than zero. This last condition is to ensure that it will have useful work to do when unblocked. If it is not possible, the first with a number of *unconsumed msgs* greater than zero is selected.
- *Affinity*: The selected process will be a process that has already ran in the processor that is asking context switching. If it doesn't exist then the first process with unconsumed messages greater or equal than zero is selected.

After applying the heuristic if there isn't any eligible process, then the context switch is not done. Notice that there will be several process migrations because the last cpu where a process was allocated will not be taken into account except for *Affinity*. However, in this case it is not a strong condition because if there aren't ready processes that satisfies the condition, then it is ignored the affinity.

Notice also that when applying global queues process mapping to processors is not a crucial decision as in Local queues, since they may be migrated during execution. Another interesting aspect of this approach is that the next process to run is chosen from the global queue, so there will be a higher probability of finding a process which can continue executing than in Local queues.

5. Our proposal: The load balancing detector (LBD)

We propose a mechanism which decides dynamically, without any previous knowledge of the application, whether to apply Local or Global process queues to it by measuring its load balancing degree.

We have observed that usually the applications at the beginning of the execution have an imbalanced behaviour, because processes start creating, the data is distributed among them, and after that, they normally perform a global synchronization function. So at this point all the jobs behave as imbalanced ones, after that they start doing regular work. As a matter of fact the coefficient of variation¹ (CV) of the number of context switches detected among processes during this period is higher than the rest of the execution due to this chaotic behaviour.

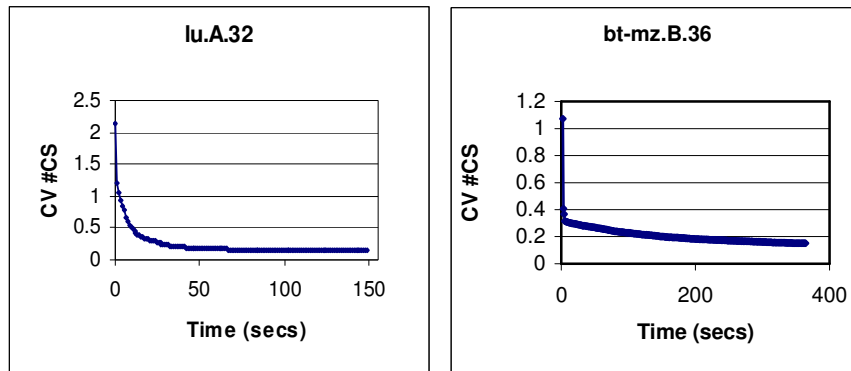


Fig. 1. Coefficient of variation of the number of context switches for a well-balanced application: lu.A and for an imbalanced application: bt-mz.B.36

In Fig. 1 it is possible to see graphically the coefficient of variation of the number of context switches (CVCS) for lu.A as a well-balanced application and for the bt-mz.B.36 as an imbalanced one. Both executed applying Global queue types for the whole running with MPL=4. After a while the CVCS goes down and remains constant until the end of the execution, no matter the load balancing degree of the jobs. We can observe also that the time spent by the job in doing the initializations is variable and depends on it. We base our criterion to measure at runtime the balance degree of an application on two CVs: the number of context switches and the user time. The first one gives us an impression when exactly the job has started executing regularly. The second one quantifies the difference in time spent in calculation by each process in order to measure their balance degree.

The user time per process shows exactly the time spent in calculation. By calculating the coefficient of variation of the user time (CVUT) among processes from a job, it is possible to quantify its load balancing degree. During the calculation of this measure we apply Global queue type to the job, to ensure a fair distribution of

¹ Percentage of the standard deviation in the average value.

the cpu time among processes, until a decision is made. As the beginning of the execution is chaotic, we delay any decision until this phase has finished.

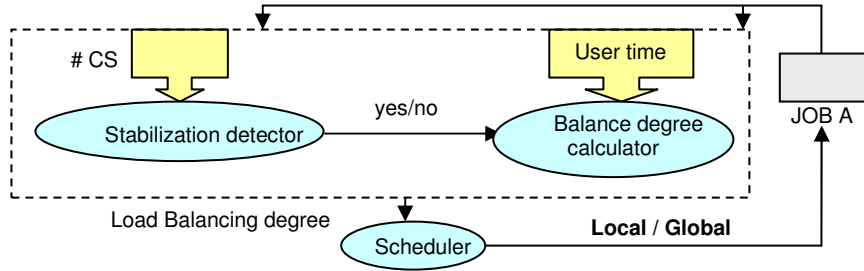


Fig. 2. Internal structure of the CPUM with the mechanism proposed to classify jobs depending on their load balancing degree

In Fig. 2 we can see graphically the internal structure of the the CPUM, used for the implementation of the mechanism. We have the *Stabilization detector* which is in charge of calculating the CVCS given the number of context switches from the processes of a job. This detector determines when the job has passed the initialization part by analyzing the CVCS and stating when it has reached a constant value. After that, the detector informs the *Balance degree calculator* that the CVUT is now a valid indicator. This calculator has also the information about the user time of each process from a job to calculate the CVUT. Finally the scheduler determines with the CVUT if the job must be considered balanced or imbalanced.

Once the job is classified the scheduler decides whether to continue the execution using Global queues or to switch to Local queues.

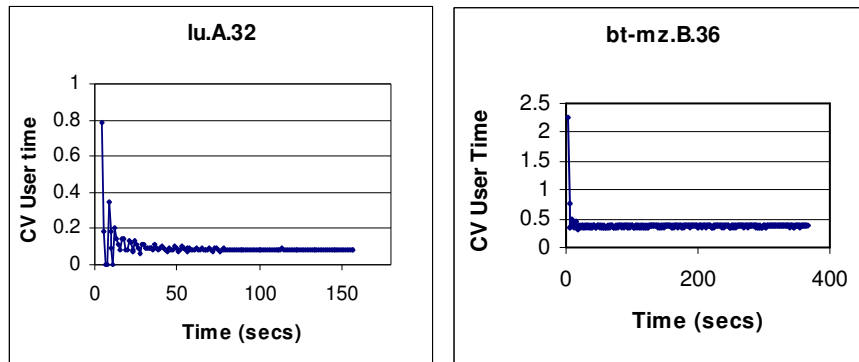


Fig. 3. Coefficient of variation of the user time for two well-balanced applications: cg.B and lu.A and two imbalanced applications: a synthetic u.50x1 and the bt-mz.B.36.

In Fig. 3, we show graphically the CVUTs for the same executions showed in Fig. 1. It can be viewed clearly that at the beginning of the execution the CVUTs are high,

and then they go down depending on the load balancing degree of the job. This means that the current CVUT can give us a true idea of the balance degree of the job.

6 Evaluations

We conducted the experiments in the following way: first we ran isolated all the applications with MPL=2, 4 and 6, applying Global queues described in Section 4.2 and comparing with the performance under Local queues. From this part we take the heuristic that showed the best performance. In the second part of the evaluation we re-execute all the applications with MPL=4 under Local and Global queues statically and using our LDB described in Section 5.

In this section we present the characteristics of the platform used for the implementations and after that we describe the applications used for the evaluations.

6.1 Architecture

Our implementation was done on a CC-NUMA shared memory multiprocessor, the SGI Origin 2000 [16]. It has 64 processors, organized in 32 nodes with two 250MHZ MIPS R10000 processors each. The machine has 16 Gb of main memory of nodes (512 Mb per node) with a page size of 16 Kbytes. Each pair of nodes is connected to a network router. The operating system where we have worked is IRIX version is 6.5 and on its native MPI library with no modifications.

6.2 Applications and workloads

For the evaluations we use as well-balanced applications the NAS Benchmarks: cg, lu, bt, sp, ep, ft and as imbalanced applications the bt-mz multi-zone version [27] and synthetic applications. They consist of a loop with three phases: messages, calculation, and a barrier. The amount of calculation varies in order to generate imbalance. We use two types of imbalance: a) 50% of processes with 2, 3 and 6 times greater in calculation than the other 50%, see Fig. 4; b) a random number of amount of calculation for each process with the biggest 6 times greater than the smallest.

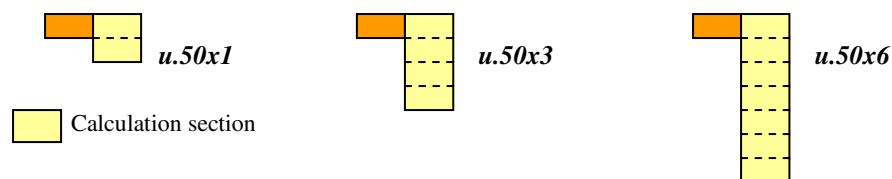


Fig. 4. Different imbalance degree changing the amount of calculation.

We worked with a number of processes per application of 32, running in isolation on a processor partition sized depending on the MPL. For the first part of the evaluation we used MPLs=2, 4 and 6, so the processor partition sized 16, 8 and 5 respectively. For the second part we executed with MPL=4 so the partition size was 8.

6.3 Performance results

First we compare heuristics to choose the next to run from a global queue, and in the second part we evaluate the performance of the LDB.

6.3.1 Global queues

Here we present the performance results for the applications with different MPLs under Global queues schemes, with all the heuristics described before.

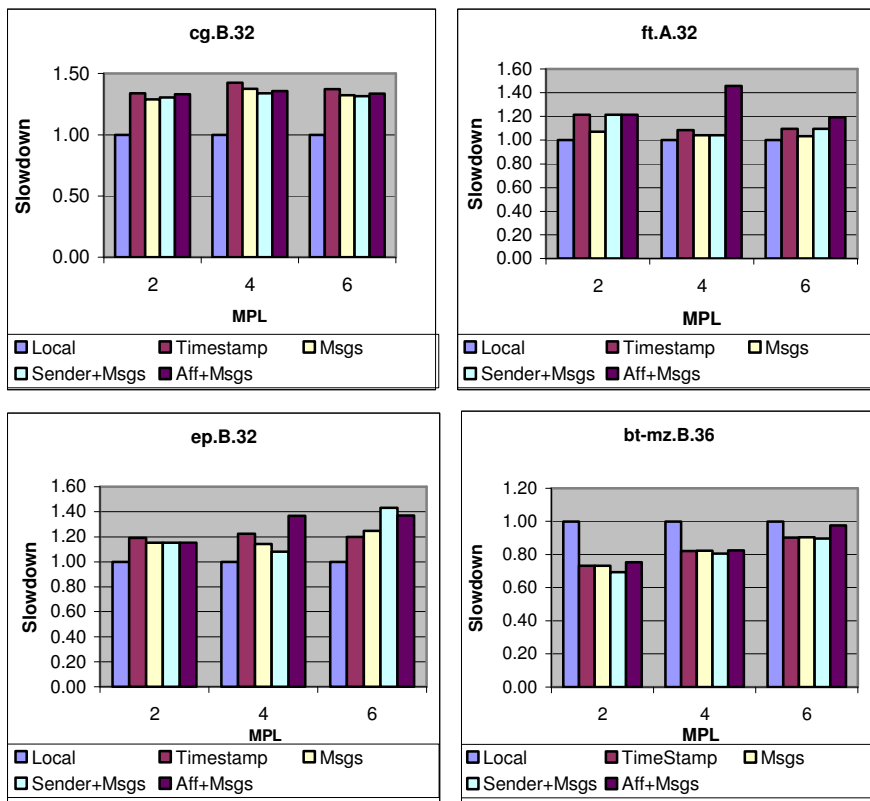


Fig. 5. Slowdown of the execution times under Global queues with respect to Local queues varying the # assigned processes per processor (MPL).

6.3.1.2 Well-balanced applications

For the well-balanced case we present a high communication degree applications, the cg.B, one with global synchronizations, the ft.A and one low communication degree, the ep.B.

We can see in Fig. 5, the slowdowns with respect to Local queues of well-balanced applications cg.B.32, the ft.A.32 and the ep.B.32 evaluated using global queues with

the different heuristics to choose the next to run. For the *cg.B.32*, the *Sender+Msgs* heuristic seem to work best. The worst is the *Timestamp* which does not take into account the synchronization between processes, which is an important issue because the *cg* is high communication degree.

The main difference between local and global approximations is the number of process migrations. While under Local queues this number is zero, under Global queues it increases significantly, thus generating context switching overhead effect degrading performance.

On the other hand the number of times a process asks context switching diminishes under Global queues because it has a higher chance of choosing the “right” process to run than Local queues, incrementing the number of coscheduled processes.

We conclude that for well-balanced jobs, even Global queues reduce the number of context switches, they don’t compensate the locality provided by Local queues.

The *ft.A*, in spite of being well-balanced performs similar under the Local and Global approaches. This is may be due to it is composed only by global synchronizations, so their processes must executed as much as coscheduled as possible. While locality favours the local approach, the Global approach ensures a more fair distribution of the cpu time among processes.

For the *ep.B*, the differences are diminished because the processes does not block very often as they rarely call blocking functions.

6.3.1.2 Imbalanced application

For the imbalanced case, we show in Fig. 5 the *bt-mz*, a multi-zone version from the NAS Benchmarks. Here a Global queue seems a more attractive option than using Local queues as the slowdown is under 1. This is due to the automatic load balancing effect. We can observe that for *Sender+Msgs* heuristics the coefficient of context switches over the execution time is reduced with respect to Local queues in about 20 % for $MPL=4$.

In conclusion, analyzing the performance of the jobs showed in the figures above the best option for well-balanced high and low communication degree applications is Local queues, as they have a slowdown greater than 1. On the other hand the imbalanced synthetic application, using Global queues seems a more attractive option. The *FT* and *EP* seems to work well under both schemes, global and local. About the heuristics when using Global queues, select the next process to run the one with the greater number of unconsumed messages seems the best option.

For the rest of the evaluation we use $MPL=4$, as it is the maximum MPL that can be reached before performance degrades [24].

6.3.2 Evaluating our proposal: LDB

In this section we present the performance results of the applications using local, global and dynamic queues. Notice that the execution time under the dynamic queue approach is always a little worse than the best of the rest of queue types. This is caused by the initial CV calculation.

In Table 1 we show the execution times for the applications running under Local, Global and under the LDB, that is decide at runtime the appropriate queue type.

After analyzing the executions, we stated empirically that with a CVUT below 0.1 shows the application is well-balanced, otherwise is imbalanced.

The difference between both approximations are the number of migrations, while using Local queues this number is zero, keeping affinity, under Global queues increments significantly, thus generating context switching overhead effect degrading performance.

Table 1 Execution time in seconds using local, global and dynamic queues.

	Local	Global	LDB	Avg CVUT
bt.A.36	298	342	306	0.02
cg.B.32	390	521	408	0.025
sp.B.36	193	248	195	0.032
lu.A.32	135	151	140	0.05
mg.B.32	54	80	58	0.071
ft.A.32	23	24	25	0.139
ep.B.32	49	51	51	1.128
u.50x1	290	236	236	0.316
u.50x3	437	315	315	0.482
u.50x6	825	538	538	0.686
u.rand.1	1307	1152	1152	0.14
u.rand.2	1287	1111	1111	0.217
u.rand.3	1279	1046	1046	0.322
bt-mz.B.36	447	368	368	0.386

On the other hand the number of times a process asks for context switching diminishes under Global queues, because a process has a higher chance of choosing the right process to run than in Local queues. In this way the number of coscheduled processes, is increased.

Finally we conclude that in spite of Global queues reduce the number of context switches really taken, they don't compensate the locality provided by the Local queues which obtain the best performance.

7 Conclusions and future work

Given the many-to-few relation of processes to processors allocated to a job, the question is whether to map processes to processors and then use a local queue on each one, or to have all the processors share a single global queue. Without shared memory a global queue is difficult to implement efficiently.

In this work we propose a mechanism, the Load Balancing Detector (LDB), to classify applications dynamically, without any previous knowledge of it, depending on their balance degree and apply the appropriate process queue type to each job.

The work consisted on two parts. First we analyze several heuristics to apply in Global queues to choose the next to run. We founded that the sender process of the

currently running, performs best. In the second part we evaluate our proposal, the LDB. We use the NAS benchmarks and several synthetic applications.

Our proposal demonstrated to work quite well, especially for the imbalanced jobs. The well-balanced jobs suffer from an overhead which is not crucial; they still work better under the new scheme than under the Global one. However for the FT and EP as they behave as well-balanced jobs, they switched queue type suffering from an overhead, having the worst performance under our proposal. On the other hand, our mechanism applies to each job independently, that means that in a workload there may be jobs executing with different queue types.

For the future we plan to extend our experiments to a wide variety of applications in order to establish a more accurate limit for the coefficient of variation of the user time (CVUT) and context switches (CVCC) among processes, to determine the balance degree of a job.

We are planning also to use the CVUT and the CVCC to map processes to processors in to implement a dynamic load balancing with Local queues.

8 Acknowledgments

This work was supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01 and the HiPEAC European Network of Excellence. And has been developed using the resources of the DAC at the UPC and the European Centre for Parallelism of Barcelona (CEPBA).

9 Bibliography

1. T.E. Anderson, E.D. Lazowska, and H.M. Levy, *The performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors*, IEEE Trans. on Comp., 38(12):1631-1644, Dec. 1989
2. A.Arpaci-Dusseau, D. Culler. *Implicit Co-Scheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*. ACM Trans. Comp. Sys. 19(3), pp.283-331, Aug. 2001.
3. D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, *The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020*, NASA, Dec 1995.
4. B.N. Bershad , E.D. Lazowska, H.M.Levy , *The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors*, IEEE Trans. on Comp., 38(12):1631-1644, Dec. 1989
5. M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. *Object-Based Adaptive Load Balancing for MPI Programs*. In Proc. of the Int. Conf. on Comp. Science, San Fr., CA, LNCS 2074, pp 108–117, May 2001.
6. D. L. Black, *Scheduling support for concurrency and parallelism in the Mach operating system*. Computer 23(5), pp. 35-43, May 1990. [16] Silicon Graphics, Inc. IRIX Admin: Resource Administration, Document number 007-3700-005, <http://techpubs.sgi.com>, 2000.
7. R. M. Bryant, H-Y Chang, and B. Rosenburg, Experience developing the RP3 operating system. Computing Systems 4(3), pp. 183-216, Summer 1991.
8. D. Feitelson. *Job Scheduling in Multiprogrammed Parallel Systems*. IBM Research Report RC 19790 (87657), October 1994. Second Revision, Aug 1997.

9. D.G. Feitelson and M.A. Jette. *Improved Utilization and Responsiveness with Gang Scheduling*. JSSPP '97, Vol 1291 of LNCS 1997.
10. E. Frachtenberg, D. Feitelson, F. Petrini, J. Fernández, Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. In IPDPS'03.
11. R. Gupta, "Synchronization and Communication Costs of Loop Partitioning on Shared-Memory Multiprocessor Systems", In ICPP'89. pp II:23-30, 1989
12. 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 Comp. Syst., pp. 120-132, May 1991.
13. F. Hofmann, M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, C. Linster, T. Thiel, S. Turowski, *MEMSY: a modular expandable multiprocessor system*. In Parallel Comp. Arch., A. Bode and M. Dal Cin (eds.), pp. 15-30, Springer Verlag, 1993. LNCS Vol. 732.
14. T. LeBlanc, M. Scott, C. Brown. *Largescale parallel programming: experience with the BBN Butterfly parallel processor*. In Proc. ACM/SIGPLAN, pp. 161-172, Jul. 1988
15. X. Martorell, J. Corbalán, D. Nikolopoulos, J. I. Navarro, E. Polychronopoulos, T. Papatheodorou, J. Labarta. *A Tool to Schedule Parallel Applications on Multiprocessors: the NANOS CPU Manager*. LNCS, 1911, pp 55-69. Springer 2000.
16. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. Int. Journal of SuperComputer Jobs, 8(3/4):165-414, 1994.
17. J.E. Moreira, W. Chan, L.L. Fong, H. Franke, M.A. Jette. *An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments*. In SC'98.
18. S. Nagar, A. Banerjee, A. Sivasubramanian, and C.R. Das. *A Closer Look at Coscheduling Approaches for a Network of Workstations*. In 11th ACM Symp. on Parallel Algorithms.
19. M. Scott, T. LeBlanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, N. Smithline, *Implementation issues for the Psyche multiprocessor operating system*. Comp. Syst. 3(1), pp. 101-137, 1990.
20. A. Serra, N. Navarro, T. Cortes, *DITools: Applicationlevel Support for oids Dynamic Extension and Flexible Composition*, in Proc. of the USENIX Annual Technical Conference, pp. 225-238, Jun. 2000.
21. M.S. Squillante and R.D. Nelson, *Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling*, In Proc. of the 1991 ACM SIGMETRICS Conf. on Measurement and Modeling of Comp. Syst., pp 143-145, May 1991
22. R. Thomas, W. Crowther, *The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors*, In Proc. of the ICPP'88, pp 245-254, Aug. 1998
23. Tucker, A. and Gupta, A. *Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*. In Proc. of the SOSP'89, pp. 159-166, Dec. 1989
24. G. Utrera, J. Corbalán, J. Labarta. *Scheduling of MPI applications: Self Co-Scheduling*. LNCS Vol. 3149, Springer 2004.
25. G. Utrera, J. Corbalán, J. Labarta. *Implementing Malleability on MPI Jobs*. In PACT'04, pp. 215-224.
26. R. Vaswani, J. Zahorjan. *Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors*, In Proc. SOSP'91 pp 26-40, Oct. 1991
27. www.nas.gov/News/Techreports/2003/PDF/nas-03-010.pdf