Predictive Coscheduling Implementation in a Non-dedicated Linux Cluster*

Francesc Solsona[1], Francesc Giné[1], Porfidio Hernández[2], and Emilio Luque[2]

[1] Departamento de Informática e Ingeniería Industrial, Universitat de Lleida, Spain
{francesc,sisco}@eup.udl.es
[2] Departamento de Informática, Universitat Autònoma de Barcelona, Spain
{p.hernandez,e.luque}@cc.uab.es

Abstract. Our research is focussed on keeping both local and parallel jobs together in a non-dedicated cluster or NOW (Network Of Workstations) and efficiently scheduling them by means of coscheduling mechanisms.
A real implementation of a predictive coscheduling technique in a Linux cluster is presented in this article and its performance analyzed and compared with other coscheduling algorithms in the literature.

1 Introduction

The studies in [1] indicate that the workstations in a non-dedicated cluster or NOW are normally underloaded. There are basically two methods of making use of these CPU idle cycles, namely task migration [3] and job scheduling [5,8]. In a NOW, in accordance with the research carried out by Arpaci [6], task migration overheads and the unpredictable behavior of local users may lower the effectiveness of this method.

A large number of scheduling schemes have been proposed for parallel machines. One of these is gang scheduling [2], successfully implemented in the Connection Machine CM-5, SGI workstations and so on. In gang scheduling, all the threads in a job are scheduled and de-scheduled at the same time, so threads making up jobs should be known in advance. In distributed systems like clusters or NOWs, this information is very difficult to obtain. The alternative is to identify them during execution [4]. Thus, only a sub-set of the processes are scheduled together, leading to coscheduling rather than gang scheduling.

Coscheduling deals with minimizing synchronization/communication waiting time between remote processes. Thus, coscheduling may be applied to reduce message waiting time and to make good use of the idle CPU cycles by executing distributed applications in a cluster or NOW system. Some of the relevant coscheduling work is explained below, attention being focussed on real implementations.

Explicit coscheduling [5] ensures that a simultaneous global context switch is performed in all the processors. In [12], a real explicit coscheduling algorithm was implemented in a Linux cluster. Despite the speedup achieved in

* This work was supported by the CICYT under contract TIC98-0433

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 732–742, 2001.

intensive message-passing distributed applications, the response time of the local workload slowed down significantly. Consequently, a mechanism to detect high-communicating distributed applications was incorporated into the original explicit coscheduling technique. This added more overhead in managing the overall system and detecting advantageous situations where this technique can be applied. Due to the centralized nature of this technique, fault tolerance is a problem: the possibility of master crashes or abnormal behavior of the explicit scheme should be taken into account.

In [9], implicit coscheduling ([6,7,8,9]) was implemented in an MPI environment, achieving performance for various coarse-grain message-passing distributed applications. In implicit coscheduling, a process waiting for messages spins for a determined time before blocking. In [13], a variation of implicit coscheduling was also implemented and evaluated in a Linux cluster. Even for low spin values, the spinning gain in blocked receives due to context switch reduction was exceeded by the overhead introduced in active waiting for messages. Also, the penalty in both return and response time added in local tasks generally yields poor performance. The conclusion is that active waiting for an event to occur (in our case blocking receive) is not a good solution in time sharing systems.

Demand-Based (divided between dynamic and predictive) coscheduling was first introduced in [10]. In contrast to implicit coscheduling, dynamic coscheduling deals with all message arrivals (not just those directed to blocked tasks). It is based on increasing the receiving task priority, even causing CPU preemption of the task being executed inside. It also provides a mechanism for avoiding local task starvation. In [11], despite the good behavior of this technique in a real system, only the execution of one distributed application was evaluated. Predictive coscheduling is based on scheduling the correspondents -the most recent communicated- processes in the overall system at the same time. Apart from this definition, there is no other predictive coscheduling work in the literature. It thus remains an open question.

The implementation of a predictive coscheduling algorithm in a non-dedicated cluster (the main aim of this article) implies that the sets of corresponding processes must be known in advance to schedule each of them at the same time. However, in such systems, this information is very difficult to obtain. The selection of the corresponding processes in each node is proposed, taking into account both high message communicating frequency and low penalty introduction into the delayed processes. An implicit and distributed nature is thus provided for this method and no centralized extra work need be done in managing or controlling the system, as in explicit techniques [5,12].

Furthermore, the obtaining of the communicating frequency is based on both message sending and receiving, not only on message receiving as in implicit and dynamic techniques [6,7,8,9,10,11]. There is perhaps no need to coschedule those processes only performing receiving messages. This is also true for sending processes. We think that processes performing both sending and receiving have

a higher potential need for coscheduling than those performing only sending or receiving. This way, an approximation to predictive coscheduling is performed.

2 The Linux Scheduler and Design Decisions

In this section, the Linux scheduler behavior is explained. Based on this, some predictive coscheduling design decisions are also commented on.

In Linux, the Ready to run task Queue (RQ) is implemented by a double linked list of *task_struct* structures, the Linux PCB's (Process Control Block). The fields used in implementing predictive coscheduling are:

- *policy*: scheduling policy. There are four scheduling policies in Linux. The "normal" tasks policy and three "real-time" scheduling policies (with more scheduling priority than the normal ones).
- *rt_priority*: scheduling priority between real-time tasks.
- *priority* -"*static*" priority-: scheduling priority between normal tasks. This ranges from 1 (low priority) to 40 (high priority). Default value = 20.
- *counter*: "*dynamic*" normal tasks priority. The initial value is set to the *priority* one. When the task is executing, each tick[1], *counter* is decremented towards 0 in one unit, then the CPU is yielded. Thus, the maximum *time slice* for a normal task with a default static priority is 210 ms.
- *files*: open files structure. This saves information about the task open files.
- ***freq***: sending and receiving message frequency. Added field to the structure and used later for implementing predictive coscheduling (see sec. 3). The initial value is 0.

There are other fields like *pid* (process identifier), *state* and so on, but these have no influence on our coscheduling scheme and so there are no more comments about them (see [16] or the Linux source for more information).

Tasks making up distributed applications are normally executed in a cluster as local (owner or interactive) ones, so no explicit information is supposed in advance for differentiating between both kinds of task. This way, all these tasks will have a "normal" scheduling policy. In Linux, the normal tasks have a Round Robin scheduling policy, with a variable time slice. Real-time tasks must acquire this condition explicitly, so we are not interested in, for example, promoting distributed tasks to real-time. Field *rt_priority* will have no influence on our coscheduling implementation and only one policy (the normal one) will be taken into account. However, a means to promote distributed tasks must be provided to implement predictive coscheduling in the normal task queue. As will be seen below, field *priority* and *counter* for "normal" tasks (in field *policy*) jointly with a new one (*freq*), will be used to do so.

In normal task creation, the field *counter* is set equal to the field *priority* and then it is appended to the RQ. The Linux scheduler picks up the next process to be executed (*current*) by means of the internal function *goodness*

[1] 1 tick $\simeq$ 10 ms

Algorithm 1 Linux scheduler algorithm

```
Step 1: While (RQ is empty) do skip  // the CPU is idle
Step 2: Lock RQ  // the scheduling must be performed in an exclusive manner
Step 3: Schedule current ≡ task with the highest goodness in the RQ
Step 4: Unlock RQ                  // exclusive access (to the RQ) deactivation
Step 5: Dispatch current           // execution in the CPU
Step 6: Do current accounting      // the accounting fields are updated accordingly
Step 7: Goto Step 1
```

(Algorithm 1, Step 3), which depends on the *counter* and *priority* values for normal tasks and field *rt_priority* for real-time tasks. Higher values indicate higher priority of execution, that is, the task with the highest returned value from the *goodness* function is scheduled. If all the returned RQ tasks values are 0, the field *counter* of every normal task is reset to be equal to *priority* and the scheduling process begins again. Note that to implement predictive coscheduling in an implicit manner, it is only necessary to increase the *goodness* value for the normal tasks proportionally to both receiving and sending communicating frequency (as mentioned earlier in section 1).

The socket packet-buffering queues, in the Linux system layer, have been chosen to collect message sending/receiving frequency information. Accordingly, packets would be a more accurate terminology for messages. The reason for doing so in the kernel space is that, coscheduling can thus be applied to any distributed application. For example, PVM ([17]) uses the sockets. If coscheduling were implemented in the user space, in the PVM sending or receiving libraries, no means for promoting the correspondents would be possible inside PVM (in the user space) when potential coscheduling was met. A new daemon or system call should be added, leading to extra overhead that could reduce coscheduling performance. Implementing coscheduling in the kernel space provides transparency to the overall system and allows the application of coscheduling independently of the message passing environment (PVM, MPI, etc ...). The drawback is in portability. A patch must be introduced into each cluster node in the Linux source. However, the Linux modifications are minimum.

3 Implementing a Predictive Coscheduling Algorithm

The proposed predictive algorithm (Algorithm 2) is based on the assumption that high receive-send message frequencies imply that potential coscheduling with remote processes is met and an accurate selection of the correspondents in each node can be made. Thus, a predictive coscheduling technique could be implemented by increasing the priority of the correspondents. The algorithm is based on the task frequency ($freq$) in sending and receiving messages, defined as:

$$freq = P * freq + (1 - P) * cur\_freq, \tag{1}$$

where P is the percentage assigned to the past frequency ($freq$) and $(1 - P)$ is the current frequency percentage (cur_freq), which is also defined as:

$$cur\_freq = rq + sq, \tag{2}$$

where rq (sq) is the number of packets in the receive (send) socket packet-buffering queue.

Algorithm 2 Predictive algorithm. Inside function *goodness*(*task*)

```
1 if (task-¿policy != "normal") weight = task-¿rt_priority + 1000;
2 else {
3       weight = task-¿counter;
4       if (! weight) {
5           weight += task-¿priority;
6           task-¿freq = freq( );   // freq( ) = P * task -¿freq + (1 − P) * cur_freq
7           weight += task-¿freq; }}
8 return weight;
```

In Algorithm 1, Step 3, the task with the highest *goodness* is scheduled. Algorithm 2 shows how the *goodness* value (*weight*) for a task is obtained. In line 1, the *weight* for a real-time task is obtained with the help of the *policy* and *rt_priority* task fields. The returned value for the normal tasks is *weight* = *counter* + *priority* (lines 3 to 5).

To implement predictive coscheduling, only lines 6 and 7 are added to the original goodness algorithm. The implemented function "$freq( )$" returns the communicating frequency computed as in formulas (1) and (2), and is used to increase the priority of distributed tasks. The frequency value obtained is saved in the PCB *freq* field to compute the following values for the frequency as past frequency. The current frequency (*cur_freq*) is obtained by the function *number_packets* (explained below). Initially, the first four values for *task-¿freq* are the mean of the obtained *cur_freq* values. This way a more approximated initial value for the communicating frequency is obtained.

Starvation of local tasks is avoided because when the *counter* field of any normal task reaches 0, it can not be executed while there are other tasks with non zero values in their respective *counter* fields.

Finally, it only remains to explain how the current frequency is obtained. To do so, the send and receiving socket packet-buffering queues must be accessed. The implemented function *number_packets*(*task*) (Algorithm 3) returns the number of received/transmitted packets by a task and used as *cur_freq* in line 6, Algorithm 2.

In Unix (Linux), the sockets are treated as files. Thus, it will first be necessary to identify the open files that correspond to sockets. To do so, the structure that represents each file in Linux (the *inode*) must be accessed. As Fig. 1 shows, it will be necessary to descend through the following structures in this order:

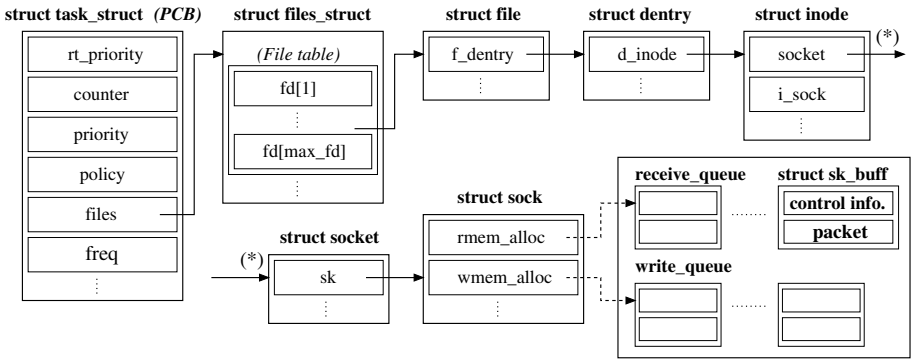

Fig. 1. Main used Linux structures (and its fields)

Algorithm 3 Function number_packets(task)

```
1  rq=rs=0;
2  if (task-¿files) {
3      for (fd = 0; fd ¡ task-¿files-¿max_fds; fd++) {
4          file = task-¿files-¿fd[fd];
5          if (file) {
6              inode = file-¿f_dentry-¿d_inode;
7              if (inode && inode-¿i_sock && (socket = socki_lookup(inode))) {
8                  rq += ⌈ socket-¿sk-¿rmem_alloc.counter / 4096 ⌉;
9                  sq += ⌈ socket-¿sk-¿wmem_alloc.counter / 4096 ⌉; }}}}
10 return (rq + sq);
```

task_struct, *files_struct*, *file*, *dentry*, and finally *inode*. Then, if the inode corresponds to a socket (condition "inode-¿i_sock" of the Algorithm 3) it will be necessary to access the socket related structures *(socket* and *sock*) by means of the obtained pointer to the socket, socket = *socki_lookup*(inode), where *socki_lookup* is an internal Linux function.

The *sock* structure points to two lists of *sk_buf* structures, which buffer packets (the socket transmission unit, = 4096 Bytes or 4KB). One such list, called the *receive_queue*, buffers receiving packets, and the other, called the *write_queue*, buffers packets to be transmitted. The *rmem_alloc* sock field saves the number of bytes in *receive_queue*, and the *wmem_alloc* field saves the number of bytes in the *write_queue*. The returned value for this function is then assigned to the current frequency (*cur_freq*).

4 Experimentation

The experimental environment used in this study was composed of four 350MHz Pentium II with 128MB of memory and 512KB of cache. They were all connected through a 100Mbps bandwidth Ethernet network and a minimal latency

in the order of 0.1 ms. The performance of the coscheduling implementation was evaluated by running *IS* and *MG*, two PVM distributed applications from the NAS parallel benchmarks suite ([14],[15]) and another synthetic one (called *master-slave*).

Three different environments were evaluated and compared between them, the plain Linux scheduler, (denoted as LINUX), Predictive coscheduling (denoted as PRED) and Dynamic coscheduling (denoted as DYN). DYN is a particular case of PRED, where only receiving frequency is taken into account. That is, line 9 in Algorithm 3 is not executed.

The local or user workload characterization in each node of the cluster was carried out by means of running an application (called *calcula*) which performs floating point operations indefinitely (or a variable number of times when the local task overhead is measured). This loading is not typical in real clusters (normally, more interactive, I/O-bound and with more unpredictable behavior). In the other hand, it is more helpful in obtaining and comparing performance of the different environments.

4.1 Predictive Coscheduling Performance

Fig. 2 shows the execution time of both NAS benchmarks when the number of local tasks (instances of *calcula*) is increased. As in the rest of the Figures of this experimentation, one of the low axis (local tasks) indicates the number of local tasks in the ready queue and the other one the different models and P values for DYN and PRED. The value P=0 and P=1 represents values for P close to 0 and 1 (i.e. P$\simeq$0 and P$\simeq$1) respectively.

In general, the performance of both coscheduling models was better than the LINUX one as the multiprogramming level increased (except for the case DYN and P$\simeq$1). In both benchmarks, the best results were obtained with the predictive model (and more precisely when P$\simeq$1). This difference was due to predictive coscheduling takes both the sending and receiving messages into account, whereas dynamic only considers the receiving ones, so the opportunity for promoting through the ready queue is greater in the predictive case. The necessity for coscheduling and the communicating pattern is more closely approximated to by such a model.

The NAS benchmarks were executed simultaneously to evaluate performance in executing various distributed applications (see Fig. 3). It is important to mention that both benchmarks fit in the main memory together with the local workload. If not, the page faulting mechanism (one or two orders of magnitude slower than the network latency) would corrupt the performance results. In this case, and for the memory fitting requirement explained above, IS is class A and MG is class T (with 600 iterations). In all the rest of the experimentation, both IS and MG are class A. The difference between the different classes is that class A of every benchmark is scaled with respect to the same one in class T.

In general, better IS results were even obtained for the PRED case. It confirms the good behavior of the predictive model with respect to LINUX and DYN. The MG results are very similar because class T is not as message-passing

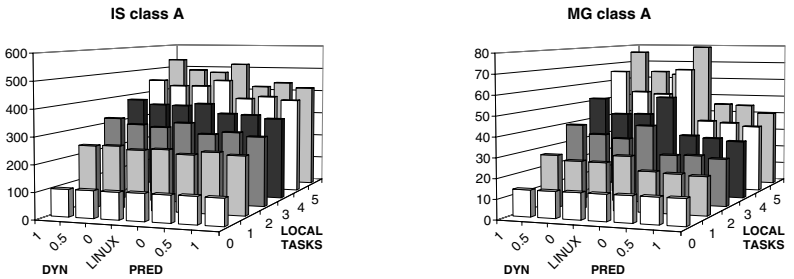

Fig. 2. NAS bench. execution times (in seconds): (left) IS, (right) MG

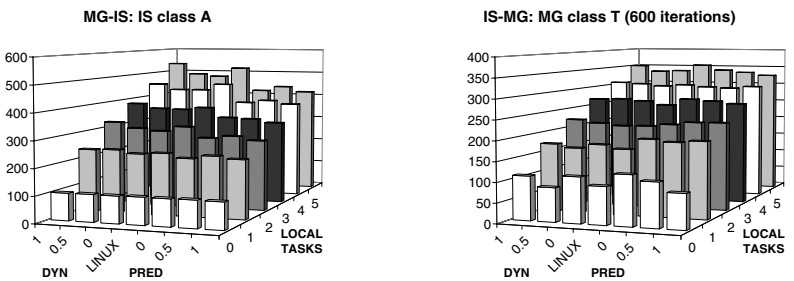

Fig. 3. Simultaneously IS and MG execution times (in seconds): (left) IS, (right) MG

intensive as class A or even as IS class A, and consequently synchronization measures provided by DYN and PRED models have fewer opportunities to improve performance.

4.2 Local Tasks Overhead and P Tuning

Fig. 4(left) shows the overhead introduced by the coscheduling models in the execution of *calcula* together with IS, MG, and both IS (class A) and MG (class T, 600 iterations).

It can be seen that the results are very similar in the three cases (with a slightly better result obtained for the LINUX model, as the communicating frequency is not taken into account). This means that this coscheduling implementation avoids the starvation of the local tasks. The reason is that the local tasks are executed when all the PCB *counter* fields of the distributed applications reach the value 0 (they have consumed their time slices). Thus, an opportunity arises for the execution of local tasks. The coscheduling methods advance the execution of distributed tasks (because it is necessary for them to be coscheduled with their correspondents) without delaying the local ones excessively.

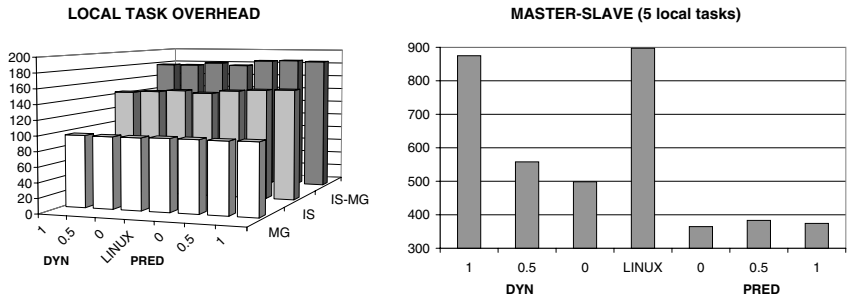

Fig. 4. Execution times (in seconds): (left) *calcula*, (right) *master-slave*

Finally, a synthetic distributed application (*master-slave*) was developed. It illustrates a case where the P parameter has more importance than in the previous ones. The *master-slave* application is made up of one master and six slave tasks. There are also two kinds of slaves, one that only receives messages (*r_slave*) and other that performs both receiving and sending messages (*rs_slave*). The mapping in 4 nodes is performed as follows: the master is assigned to one node and two slaves (of different kinds) to each remaining node. The master performs a predetermined number of iterations and then, both master and the slaves finish execution and the master reports the return time for the application. In each iteration, the master sends a message to all the slaves. All the slaves, after receiving the message, perform a simple floating-point computation. In addition, the *rs_slaves* reply to the master with the computation result. After receiving all the *rs_slave* messages, the master repeats the process again.

Fig. 4(right) shows the good performance of the predictive model in the execution of the synthetic application with a workload of five local tasks in each node. As was expected, PRED promoted the *rs_slave* tasks earlier than DYN and LINUX, so the round-trip time of each iteration decreased. Moreover, P$\simeq$0 obtained the best results because taking the current receiving frequency into account, a closer approximation to the message pattern was obtained. However, any mean between the current and past receiving frequency caused a drop in performance. Note as P has more influence in the DYN case: performance decreases strongly by increasing P.

5 Conclusions and Future Work

A predictive coscheduling technique with reasonable performance was implemented in a Linux cluster and discussed and compared with dynamic coscheduling and the plain Linux scheduler. The experimental results obtained corroborated the importance of applying a coscheduling technique over a non-dedicated cluster and the predictive model in particular.

Future work is directed towards proposing more coscheduling techniques taking into account network latency, paging faults, context switch costs, etc... More-

over, as was shown, the coscheduling performance may vary depending on P, the message pattern of distributed applications and their relationship. Consequently, a more accurate analysis based on this should be performed.

The future trend is to determine metrics for evaluating the effects of coscheduling techniques on execution time. They should be employed this way in tuning parameters while the distributed tasks are executing and not in later executions. The traditional ones (speedup, efficiency, etc...) only serve for performance evaluation at execution end.

References

1. Anderson, T., Culler, D., Patterson, D. and the Now team: A case for NOW (Networks of Workstations). IEEE Micro. 1995. 732
2. Ousterhout, J. K.: Scheduling Techniques for Concurrent Systems. 3rd International Conference on Distributed Computing Systems. 1982. 732
3. Litzkow, M., Livny, M. and Mutka, M.: Condor - A Hunter of Idle Workstations. 8th Int'l Conference of Distributed Computing Systems. 1988. 732
4. Feitelson, D. G. and Rudolph, L.: Coscheduling Based on Runtime Identification of Activity Working Sets. International J. Parallel Programming 23 (2). 1995. 732
5. Crovella, M. et al.: Multiprogramming on Multiprocessors. 3rd IEEE Symposium on Parallel and Distributed Processing. 1994. 732, 733
6. Arpaci, R. H., Dusseau, A. C., Vahdat, A. M., Liu, L. T., Anderson, T. E. and Patterson, D. A.: The Interaction of Parallel and Sequential Workloads on a Network of Workstations. ACM SIGMETRICS'95. 1995. 732, 733
7. Arpaci, R. H., Dusseau, A. C., Culler, D. E. and Mainwaring, A. M.: Scheduling with Implicit Information in Distributed Systems. ACM SIGMETRICS'98. 1998. 733
8. Dusseau, A. C., Arpaci, R. H. and Culler, D. E.: Effective Distributed Scheduling of Parallel Workloads. ACM SIGMETRICS'96. 1996. 732, 733
9. Wong, F. C., Arpaci-Dusseau, A. C. and Culler, D. E.: Building MPI for Multiprogramming Systems Using Implicit Information. 6th European PVM/MPI User's Group Meeting. LNCS. 1999. 733
10. Sobalvarro, P. G. and Weihl, W. E..: Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing. 1995. 733
11. Sobalvarro, P. G., Pakin, S., Weihl, W. E. and Chien, A. A.: Dynamic Coscheduling on Workstation Clusters. IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing. 1998. 733
12. Solsona, F., Giné, F., Molina, F., Hernández, P. and Luque, E.: Implementing and Analysing an Effective Explicit Coscheduling Algorithm on a NOW. VECPAR'2000. LNCS vol. 1981. 2001. 732, 733
13. Solsona, F., Giné, F., Hernández, P. and Luque, E.: Implementing Explicit and Implicit Coscheduling in a PVM Environment. Europar'2000. LNCS vol. 1900. 2000. 733
14. Bailey, D. et al.: The NAS parallel benchmarks. International Journal of Supercomputer Applications 5 (3). 1991. 738
15. Parkbench Committe: Parkbench 2.0. http://www.netlib.org/parkbench. 1996. 738

16. Beck, M., et al.: LINUX Kernel Internals. Addison-Wesley. 1996. 734
17. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.: PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. MIT Press. 1994. 735