Load Balancing Based on Process Migration for MPI

Georg Stellner and Jörg Trinitis

Institut für Informatik der Technischen Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München
{stellner,trinitis}@informatik.tu-muenchen.de
http://wwwbode.informatik.tu-muenchen.de/~stellner
Tel.: ++49-89-28925348, Fax: ++49-89-28928232

Abstract. Process migration is one technique to implement environments that perform automatic load balancing. However on networks of workstations the load indices and heuristics that are used must respect the load that is imposed on the system by other users' processes. In this paper we suggest an approach that uses an existing process migration component to construct an automatic load balancing system for MPI applications. Both the load indices and the heuristics consider load that is imposed on the system due to other users activity. For a computational fluid dynamics application performance improvements between 10% and 54% could be achieved.

1 Consistent Checkpointing with CoCheck

The CoCheck environment allows both the creation of checkpoints and the migration of processes of parallel applications on networks of workstations. Initially CoCheck extended PVM [3], so that PVM applications could be started under the control of a resource management system [9]. In that case, CoCheck was used to create checkpoints in order to provide global scheduling of parallel applications. Although process migration was already supported, its performance needed further improvement. Consequently, the focus of the research was set on performance improvements of checkpointing and particularly process migration. This could be achieved by transferring the checkpoints directly over TCP network connections [8]. As the next step, CoCheck was implemented to support the proposed MPI [5] message passing standard. Therefore, the protocol was integrated with tuMPI[1] which is an implementation of the MPI standard definition [10]. As could be shown in [10] migration times of a process are depended on the size of the migrated process. The time to migrate a single process is given in 1.

$$t(x) = 1.77s + \frac{x}{763kBytes/s} \tag{1}$$

[1] Technische Universität München Message Passing Interface

For that implementation of CoCheck a load balancer was added that performs automatic load balancing by process migration. The load balancer comprises a component to gather load information from all nodes. This information is used by a decision component to determine nodes that are overloaded and underloaded respectively. Among the processes on the overloaded nodes candidates have to be selected which will be migrated to new nodes. Finally, a process is selected and migrated.

The remainder of this paper is organized as follows. Firstly, an overview on related work is presented. Then the automatic load balancing system for MPI is explained. After that performance results of the load balancing system will be discussed. Finally a conclusion and an outlook on future work is given.

2 Related Work

Andres *et al.* [1] describe an environment that performs automatic load balancing on a network of workstations for PVM applications. In their approach the migration component is completely integrated into the PVM system. On each node a load monitor determines the current load. This information is broadcasted to all PVM daemons. Each daemon sets up a matrix with the current load indices of all the nodes. Upon request this matrix is made available to the load balancer. Migrations are only performed if the load imbalance exceeds a predefined constant which represents the cost of the migration. Processes which dominate the current processor or which perform only small amounts of work are not migrated. Also the number of migrations is restricted per process to avoid thrashing. In [1] Andres *et al.* conclude that "Although we did not achieve the dramatic performance improvements we had hoped for when we implemented load migration under PVM, a great deal was learnt about what makes a good or bad migration heuristic [...]."

Hector [6] provides dynamic task allocation to MPI applications. Its migration component has been inspired by former versions of CoCheck [8; 10]. Hector adds an additional process (task allocator) to each node of the system. Each task allocator is responsible to collect the load information on its node, launch processes and monitor the execution of the processes. In addition there is a master allocator that collects the load information of all slave allocators. Based on this global load view of the application the master allocator decides about migrating processes.

3 Automatic Load Balancing with CoCheck for tuMPI

tuMPI is an implementation of the MPI standard. It is primary intended for conducting research in the area of process migration and checkpointing [10]. The automatic load balancer which has been implemented for tuMPI consists of three components: a load measure component, a decision component and a migration component. The latter is provided by CoCheck [8; 10]. The remaining two components will be introduced now. Both have been added as an additional part

to the tuMPI daemon. This is a central component of every tuMPI application which is responsible for the process management.

3.1 Load Measurement

In accordance with the literature where simple load indices and strategies achieved good results [2] the automatic load balancer of tuMPI also uses a simple load index. It is based on the average length of the ready queue on a node during the last minute (*avenrun*). This value is available on every UNIX system and can easily be determined. With the `rup` command the value of *avenrun* can also be determined on remote nodes, so that there is no need for additional monitor processes on the nodes which execute processes of parallel applications.

At startup time of an application the user can specify two additional parameters which configure the intervals at which the load should be determined. The first parameter specifies the time between two successive measurements when no migration was necessary. The second parameter determines the time at which the next measurement is performed when a process was migrated. This facility has been introduced to allow more time after a migration for the load values to stabilize. This avoids too many migrations to a machine before the additional load of a process has shown its effects also in the *averun* value. Since the complete parameters with which an application is started are given to the call of `MPI_Init` those two parameters are removed from the parameter list during that function. Hence, the application does not need to be changed to handle the additional parameters: `MPI_Init` provides the desired transparency.

3.2 Decision Component

Although the migration of processes with CoCheck is only possible between migration compatible (homogeneous) machines[2] the decision component supports heterogeneous networks of workstations. Therefore the network is divided in homogeneous sub-clusters and the decision component tries to evenly distribute the load within each sub-cluster. As even within homogeneous sub-clusters the potential computational power of the machines can vary due to different clock frequencies, main memory capacity, etc. the above mentioned *avenrun* value of each machine is normalized with a machine specific architectural constant α. In the current implementation the user is responsible to assign this constant to each machine in the mapping table that specifies the cluster and that is processed by tuMPI during startup. Future versions will automatically assign this value to each machine. The load index that is used in the decision component is provided in (2).

[2] Migration between binary compatible machines is not always possible due to different run-time properties of processes. In the case of Sun machines the binaries can be run on any machine, but it is not possible to migrate a process from a sun4m to a sun4c implementation of the SPARC specification due to a different run-time stack. Hence the requirement of binary compatible machines is not sufficient.

$$load_i = \frac{1 + avenrun_i}{\alpha_i} \tag{2}$$

$$imbal = |\max_{i=0}^{n}(load_i) - \min_{i=0}^{n}(load_i)| \tag{3}$$

Increasing the *avenrun* value by one in the denominator guarantees that in case of unloaded machines ($avenrun = 0$) machines with different speeds can actually be distinguished. Hence, the decision component can choose the potentially faster machine in case a destination machine for a migration has to be determined.

The evaluation phase begins after the load on all machines has been determined. Therefore the normalized load indices of all nodes are calculated using (2) for a node i. After that the difference between the highest and lowest normalized value is calculated according to (3). If this difference exceeds a specified imbalance value a migration is considered. Otherwise no migration will be performed and the next measurement starts after the waiting time that has been specified (c.f. section 3.1). If however, the current load imbalance is greater than the specified imbalance value suitable migration candidate is determined. Currently, a simple strategy is used which selects a process of the application on the node with the highest normalized load index. Similarly the destination node is selected with a simple strategy: it is the node with the lowest normalized load index. Finally, CoCheck is requested to migrate the selected process from the source to the destination node.

As the *avenrun* value is influenced by processes which do not belong to the parallel application this approach also can cope with load imbalances caused by external influences. However problems arise when the machine pools of two tuMPI applications overlap. In this case the two load balancers might work against each other. In future versions this situation must be detected and the load balancers must coordinate their migration decisions regarding the nodes in the overlapping node set.

Currently, the above mentioned value for the load imbalance that has to be exceeded so that migrations are actually performed to level the load must be specified by the user as an additional parameter on the command line. As in the case of the measurement intervals (c.f. section 3.1) this parameter is automatically removed during `MPI_Init`.

4 Performance Experiments

To evaluate the performance of the automatic load balancer we applied it to a computational fluid dynamics application. This application is briefly described in the next section. After that the hardware environment for the experiments and the results of the experiments are presented.

4.1 NSFLEX

NSFLEX is a computational fluid dynamics application which is used to solve problems in aerodynamics [4]. More precisely, it solves two and three dimensional Navier-Stokes and Euler Equations. The speed of the flow can stretch from 0.3 to 100 Mach. The discretization was done with a finite volume method by solving the Reynolds Equations. The linearization uses the Newton method. The turbulent flow is modeled with Baldwin-Lomax and solved with a Gauß-Seidel method. Several grid topologies are supported (C-grid, O-grid and H-grid). The NSFLEX code for MPI which has been used in the experiments was initially implemented for MPICH and solves the Cast-7 problem. The problem was partitioned in such a way, that four processes were required to compute the solution. Each of the processes occupied 6840 kBytes main memory during run-time. Most of this memory (6044 kBytes) was located in the data segment.

4.2 Hardware Environment for the Experiments

All experiments have been done on five Sun Sparc 10 machines which were equipped with 32 Mbytes of main memory and ran under the SunOS 4.1.3 operating system. The machines were physically distributed over several buildings and were interconnected via the local area Ethernet network of the computer science department. The measurements have been performed during off-peak hours in the nights and on weekends to reduce the influence of other users. Despite these precautions it was not possible to completely dedicate the machines and the network for the experiments.

4.3 Results

In a first series of experiments we were concerned about how the number of migrations would influence the execution time of NSFLEX. Therefore NSFLEX was executed on four machines and a varying number of migrations were forced to the remaining machine. The average results are depicted in Fig. 1 whereas $t(n)$ in equation (4) is the result of a linear regression applied to all measurements.

$$t(n) = 330.55 + 12.83n \tag{4}$$

Although the average values and the linear regression show a linear increase of the execution times of NSFLEX a closer look at the individual measurements unveils noteworthy details. In contrast to the expectation that migrations prolongs the execution time in any case, surprisingly also reductions could be observed. This is depicted in Fig. 2.

The explanation for this effect is as follows. Although the migration itself takes a certain amount of time the experiment already describes an automatic load balancer with a random strategy where migration decisions are based on coincidence. Since the external load could not be completely eliminated, this

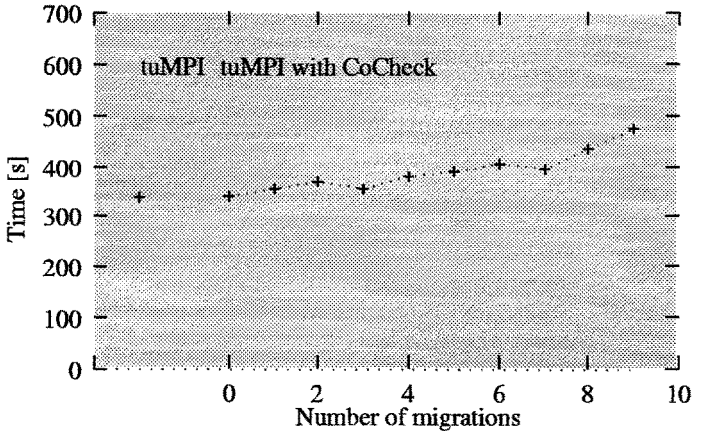

Fig. 1. Average execution times of NSFLEX depending on the number of migrations.

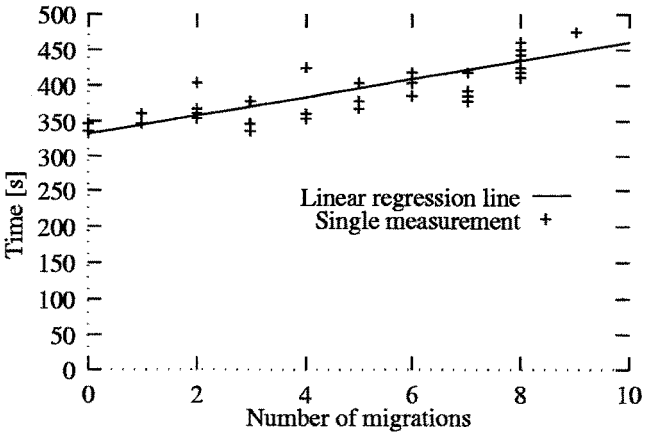

Fig. 2. Single measurements and linear regression.

simply strategy of migrating a randomly selected process lead to better load distributions under certain circumstances. The examination of the log files of the experiments, in which the load values of all machines had been recorded, showed that in the cases where better execution times could be achieved, the source nodes indeed were overloaded.

In a second series of experiments an additional process was placed on one of the nodes where an NSFLEX process was executed. The automatic load balancer was enabled and configured, so that the load imbalance value was set to 1.5 and the two interval parameters were both set to 40 s. Hence, migrations were performed if the normalized difference between the least and most loaded node exceeded 1.5 and the time between two measurements were 40 s independent of migrations having been performed or not. The additional process on the node consumed about the same amount of CPU-time as the NSFLEX process as can be seen from the output of the `top` command in Fig. 3.

```
  PID PRI SIZE  RES   STATE TIME WCPU    CPU    COMMAND
1340276  24K   144K  run   1:45 45.37%  45.31% AddLoad
1343177  6840K 2904K run   1:23 45.37%  45.31% NsFlex
```

Fig. 3. CPU time of the NSFLEX process and the additional load process.

The average execution times of NSFLEX without performing migrations increased to 697 s in comparison to 336 s without the additional load process. The experiment was repeated with the the automatic load balancer being activated. In this case the average execution time could be reduced to 327 s. In some cases more migrations were necessary to achieve an even load distribution. In these cases execution times of 374 s and 500 s respectively could be achieved (54% reduction). The individual measurements and the corresponding average values are given in Fig. 4.

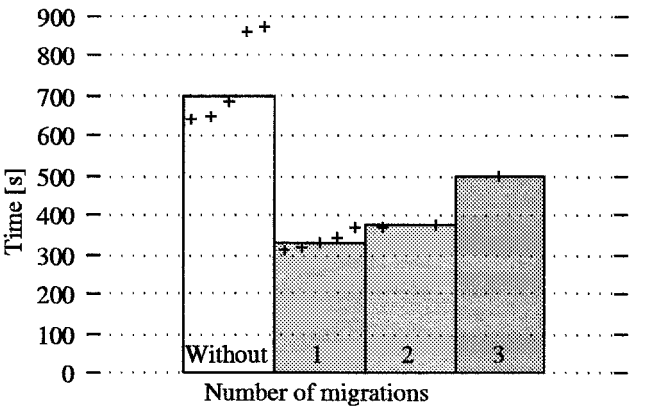

Fig. 4. Effect of load balancing on NSFLEX execution times with one additional load process.

In a similar experiment with two load processes execution times could be reduced from 636 s to at least 573 s (10% reduction).

5 Conclusion and Future Work

In concert with tuMPI CoCheck provides the basis for an automatic load balancing system for MPI applications. Both the migration times of processes (not discussed here, refer to [10] for details) and the improvements which could be

achieved using the NSFLEX application are very encouraging. Already the simple policy based on the normalized length of the ready queue and the threshold value used in the prototype implementation could reduce execution times from 10% to 54%.

Current limitations of the the decision components are that they only initiate the migration of a single process at a given time and that decision components of several applications cannot cooperate. The first limitation reduces the efficiency of distributing the load evenly whereas the latter leads to problems in case of overlapping machine pools when processes migrate to the same machine. Hence, the decision components have to be modified, so that they migrate more than one process if this is appropriate and that they can coordinate their migration decisions in case of overlapping machine pools. Furthermore, the intra-application scheduling aspect which is covered by dynamic load balancing should be extended to a resource driven inter-application scheduler which can be found in resource management systems. Finally, the heuristics must be evaluated with more applications. Particularly, the ability to balance the load in heterogeneous clusters must be examined.

References

1. D. Andres, C. Elford, B. Fin, and L. Smith. Dynamic load balancing in PVM. Tech. rep., University of Illinois at Urbanna-Champaign, April 1993.
2. D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, SE-12(5):662–675, May 1986. Abgedruckt in [7, S. 340–353].
3. A. Geist, et al. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.
4. M. Lenke, S. Rathmayer, A. Bode, T. Michl, and S. Wagner. *Parallelization with a real-world CFD application on different parallel architectures*, volume 2, chapter 8, pages 119–166. Computational Mechanics Publications, Southampton, Boston, 1995.
5. Message Passing Interface Forum. MPI: A Message Passing Interface Standard, May 1994.
6. S. Russ. Hector: Automated Task Allocation for MPI. In *Proc. of IPPS*, pages 344–348, Honolulu, HI, April 1996. IEEE CS Press, Los Alamitos, CA.
7. B. A. Shirazi, A. R. Hurson, and K. M. Kavi (eds.) *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE CS Press, Los Alamitos, CA, 1995.
8. G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. In *Proc. of the 2nd European PVM Users' Group Meeting*, pages 131–136, Lyon, September 1995. Editions Hermes.
9. G. Stellner. Consistent Checkpoints of PVM Applications. In *Proc. of the 1st European PVM Users Group Meeting*, http://www.labri.u-bordeaux.fr/~desprez/CONFS/PAPERS/abs010.ps.gz, 1994.
10. G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proc. of the IPPS*, pages 526–531, Honolulu, HI, April 1996. IEEE CS Press, Los Alamitos, CA.