Efficient Resource Management for Malleable Applications

Jan Hungershöfer, Achim Streit, and Jens-Michael Wierum

Paderborn Center for Parallel Computing
33102 Paderborn, Germany
{hunger,streit,jmwie}@upb.de

PADERBORN
CENTER FOR
PARALLEL
COMPUTING

Technical Report PC^2
TR-003-01
December 2001

Abstract. In this paper we present a method for managing concurrent parallel applications on large shared-memory machines efficiently and fair. It combines advantages of space-sharing for tight coupled parallel applications and the possibility of immediate job execution in time-sharing environments.
An *application parallelism manager* (APM) assigns system resources to running jobs on the fly. In case of changing system load, the resources are reassigned to get optimal system utilization. Policies for a fair and efficient distribution of processors to parallel applications are presented. The APM uses information about the job's scalability to maximize the efficiency of system usage.
An implementation of the APM on a 16-processor SMP has been evaluated and is presented here. It shows good results compared to an optimal offline schedule.

1 Introduction

Symmetric multiprocessor machines (SMP) in various sizes are well established both in the academia and industrial world. A SMP allows its owners and users a more flexible way of usage as the two most used parallel programming paradigms (shared-memory and message-passing) are applicable. Especially commercial applications are written applying the shared-memory paradigm because it is more convenient and less error-prone.

The scheduling of parallel jobs on large systems efficiently is still a topic of research [7, 10]. We present an efficient and comfortable way of managing malleable applications on a SMP. The approach combines short response times, efficient operation of the machine (utilisation almost always 100 %), fairness and comfortable handling.

An *application parallelism manager* (APM) determines an assignment of processors to jobs on the fly based on estimated current speedup. The applications adapt their degree of parallelism to the prescribed value. This allows an efficient distribution of the compute performance. It is assumed that applications are malleable, which means that they can change their degree of parallelism (e. g. number of threads) during execution.

One application of our industrial partners is a long running multi-threaded finite element simulation. It is used by several employees at the same time which induces overloaded computing resources. The old-fashioned and inefficient method of killing jobs manually to release some of the machine's processors needed to be replaced. In a first step the application was made malleable which solved the problem of wasted processor cycles due to jobs which were killed when they were already half-done. Inter-process-communication (IPC) signals were used to change the number of threads. But since signal communication is only applicable for jobs of the same owner, a more powerful and as well fair method was necessary that works without user-interaction.

The application and its unpredictable effect on speedup and parallel efficiency combined with the mixed machine usage for production and development makes it nearly impossible to compute an efficient static resource assignment on job submission. Especially long sequential phases which depend on the progress and input data of the job makes a static assignment more complex.

2 Background

Currently two main strategies are known in which resources can be assigned to jobs: time-sharing or space-sharing [7, 8].

Time-sharing is widely used on shared-memory machines and provides users a single-system image, so that they can use the SMP in the same way as their desktop computer. Time-sharing is usually implemented inside the operating system (OS) which distributes all running jobs across available processors. In general time-sharing is well suited for sequential jobs on a shared-memory compute server. Though it is not optimal for parallel jobs because time-sharing generally does not guarantee that coherent processes are running at the same time.

In contrast to that *space-sharing* assigns resources exclusively to jobs. The negative effects of time-sharing, e. g. cycle-stealing, are avoided. Also the phenomenon of overloading is obnoxious. Space-sharing is usually implemented in the resource-management-system on top of the OS. When a job is scheduled to start, a set of resources is assigned to the job, and this set of resources is neither changed in its size (number of elements) nor are the elements exchanged during execution.

Many difficulties arise from the concept of space-sharing and decisions from the scheduler might lead to fragmentation and free slots in the schedule for static partitions [7]. Reasons for that are, jobs do not fit together to utilize all resources or they are not started immediately in anticipation of future jobs, which might utilize resources more optimal [1, 13]. Some approaches try to estimate the runtime of future jobs based on the past [15, 16]. Results of an optimal offline schedule (submit time, runtime, and scalability are known in advance) are used as a benchmark for our approach.

A combination of time- and space-sharing is called *gang-scheduling* [4, 5]. For a specific amount of time (gang) space-sharing is used to assign (and withdraw) processors to jobs, and also jobs are scheduled simultaneously, which means that jobs only start at the beginning of a gang. Different gangs are executed in time-sharing manner. Small jobs may have to wait for the next round of their gang resulting in a lengthened response time. Due to the added flexibility of gang-scheduling the overall system-performance gets better (both response time *and* utilization) but it still does not prevent from fragmentation and free slots in the schedule. The draw-back of gang-scheduling is that changes to the OS are often necessary. This was not suitable in our case.

In Feitelson and Rudolph [6] a classification of jobs in rigid, evolving, moldable and malleable was done. In Feitelson et al. [7] *malleable* jobs are defined as jobs which number of assigned processors may change during execution as a result of the system assigning additional resources or requiring to release some. Whereas for *moldable* jobs the number of processors can only be set at start and is fixed during runtime.

3 Architecture

As already stated above the central *application parallelism manager* (APM) generates the resource assignment. The APM's policy (which metric should be optimised) is set by the system administrator. The APM is designed as a small database and two threads are used for writing and reading it to circumvent deadlocks. One thread listens on all active ports for incoming status information from the various clients (applications) and stores the information values in the database. The other thread is used to compute an adequate resource distribution and send the number of assigned resources to each connected application. Entries in the database are generated or deleted if an application connects to the server for the first time or disconnects after its completion.

The APM is implemented as a single central server due to several reasons. Tests have shown that a single server can handle a realistic number of clients. Though a distributed server might be more robust and scalable our approach has demonstrated stability and reliability in a productive environment. In case the APM is not responding the application will use its latest resource assignment. Additionally a central instance is more extensible for logging and billing mechanisms and requires less communication. The system load generated by the APM is insignificant which makes it possible to run the APM on the compute server (and not on a dedicated machine) together with all applications.

The APM and applications interchange information using standard TCP sockets. Thereby it is possible that the APM and applications are running on different machines and are also owned by distinct users. This communication approach obviously provides more flexibility than using IPC signals for communication. The data send from the application to the APM is independent from the chosen policy. The messages primarily contain the runtimes of the sequential and parallel phases of the application. The information send by the APM to the application contains only one single value specifying the number of assigned resources (i. e. threads for the application).

The information send out by the application has to be provided by the application itself. The rate of sending information by the applications is variable. First, the application reports its status periodically. Second, the application sends additional information if major changes occur. Third, the APM is able to request information from the application.

This architecture minimises the amount of necessary modifications on the application side. A small library with communication methods has to be linked to the application. As the application has to monitor the runtime of sequential and parallel phases itself, calls to monitoring functions have to be coded in the application, too. With this architecture the policy-dependent code is kept in a module inside the APM, which makes it easy to implement and use new policies. Furthermore it is assumed that all connected applications are *cooperative*, since they collect the information send to the APM by their own. Otherwise malicious applications may cheat to get more assigned resources.

The application we have used is an industrial finite element code for structural mechanics [3]. The code shows hardly predictable behaviour with respect to runtime and scalability. The main loop of the application consists of multi-threaded parallel phases and sequential parts. The time needed for the different phases depends on the input and is varying over the application runtime. Therefore the speedup is also variable. Sequential phases of the application, which occur unpredictably, may take up to thirty minutes. At the beginning of these phases the application terminates all threads except one and the processors can be assigned to other clients by the APM. Due to the characteristics of the application permanent load balancing is needed [12]. This can be done very quickly and allows rebalancing in only a few application steps if the number of assigned processors has changed.

4 Policies

The two implemented policies are equipartition (*equiP*) and maximizing the accumulated current speedup (*maxS**) on the machine. The APM has to distribute N jobs on the P processors of the compute server. Each job j gets p_j processors.

The first strategy (*equiP*) is based on a common scheduling approach. Each active job gets the same number $\bar{p}$ of processors assigned.

$$\bar{p} = \left\lfloor \frac{P}{N} \right\rfloor$$

A method for assigning the possibly remaining processors could be to distribute them evenly among the oldest jobs, for example.

$$p_j = \begin{cases} \bar{p} + 1 & , j \leq P \bmod N \\ \bar{p} & , \text{otherwise} \end{cases} \tag{1}$$

This strategy also gives some sort of fairness for the users: Each job gets the same amount of resources and smaller jobs will be finished earlier.

An alternative strategy (*maxS**) takes the scalability of jobs into account to increase the efficiency of the system usage. For this purpose we define the accumulated speedup $S^\star$ as the sum of the speedups S_j of all jobs j of the actual configuration:

$$S^\star := \sum_j S_j(p_j) \text{ , with } S_j(p_j) = \frac{T_j(1)}{T_j(p_j)}, \tag{2}$$

with $T_j(p)$ being the running time of job j using p processors. Our model maximises the accumulated speedup to achieve a maximal efficiency considering the limited number of resources: $\sum_j p_j \leq P$.

The achievable efficiency strongly depends on the quality of speedup estimations. Amdahl's Law for the parallel runtime T is applicable if the actual runtimes of all sequential and parallelised parts ($T_{\text{seq}}(p_j)$ and $T_{\text{par}}(p_j)$) are known [2].

$$T(p) = T_{\text{seq}}(p_j) + \frac{T_{\text{par}}(p_j) \cdot p_j}{p}$$

To get a more realistic model for the speedup and efficiency the overhead for the parallel processing is considered: parallel management overhead (c_1) and overhead by intra-process communication (c_2). This results in the following formula for the parallel runtime:

$$T(p) = (1 + (p-1)\,c_1) \cdot T_{\text{seq}} + (1 + p \cdot (p-1)\,c_2) \cdot \frac{T_{\text{par}}}{p} \tag{3}$$

with

$$T_{\text{seq}} = \frac{T_{\text{seq}}(p_j)}{1 + (p_j - 1)\, c_1}\ ,\ T_{\text{par}} = \frac{T_{\text{par}}(p_j) \cdot p_j}{1 + p_j \cdot (p_j - 1)\, c_2}.$$

This formulation allows an estimation of the runtime, speedup, and efficiency for each degree of parallelism based on measurements for p_j. The server can use this information to determine an actual partitioning with optimal speedup at each point in time. The parameters c_1 and c_2 have to be determined for each machine/application combination. This includes the machine characteristics latency and bandwidth for messages and synchronisation as well as the management and communication requirements of the application. The parameters can be determined by interpolation of scalability curves of some simple test runs.

Due to the fact that our model describes a monotonous decreasing efficiency, we get a simple distribution scheme for scattering the processors among the clients. Starting with one processor for each client, the server determines the *differential speedup* $DS(p)$ for each client. $DS(p)$ is the speedup difference of a client when going from $p-1$ to p processors. The client with the largest differential speedup gets a processor assigned. The new differential speedup $DS(p)$ is calculated for that client and the process is continued until all processors are distributed.

Fig. 1 shows in a small artificial example the advantages for scheduling using malleable applications. Three jobs (A, B, C) starting at time t_j have to be scheduled on a 4-processor machine. The sequential and parallel running times with one processor are T_{seq} and $T_{\text{par}}(1)$. $DS(p)$ denotes the differential speedups. The jobs can be done after 12.16 seconds when they are malleable and managed with the above mentioned scheme. Otherwise – for non-malleable but moldable jobs – 14 seconds are needed to execute them.

	t_j	T_{seq}	$T_{\text{par}}(1)$	$DS(p)$			
	[s]	[s]	[s]	1	2	3	4
A	0	2	12	1.00	0.75	0.58	0.47
B	1	1	24	1.00	0.92	0.86	0.79
C	3	1	6	1.00	0.75	0.58	0.47

A 🟩 B 🟥 C 🟦

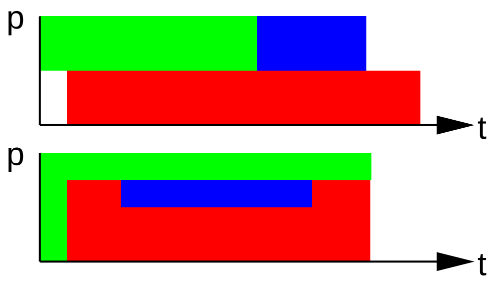

Fig. 1. Optimal offline scheduling with moldable jobs (top) versus dynamic online scheduling with malleable jobs using the *maxS** policy (bottom).

5 Results

The server has been tested on a 16-processor SMP system with a set of different scenarios. Each scenario contains a set of jobs with variations in scalability and runtime. The results indicate that the behaviour of both implemented policies get more similar as the number of jobs on the system increases. This is due to the fact that most of our jobs scale quite well for a small number of threads. An equal distribution of processors to jobs gives an accumulated speedup comparable to the second policy. In these cases the offline scheduling often performs better than the scheduling by the APM, because it has many possibilities to schedule the jobs with moderate degree of parallelism reaching a high throughput.

The equipartition policy leads to better response times while the speedup oriented approach increases the throughput and therefore the efficiency of the system's usage.

Scheduling many malleable jobs on a large system leads to a complex behaviour which is hard to understand from a small figure. We discuss a small scenario with just four jobs on eight processors for better visualisation. Table 1 describes the used scenario. The jobs A to D are submitted at distinct times and are quite different in terms of runtime and scalability.

Fig. 2 shows the resulting schedules. Note that due to the different scalability of the jobs the area of the determined partitions depend on the height of the partition (e. g. job D). Furthermore, the y-axis represents only the number of assigned processors, but not the processors itself, e. g. job B of the top most graph does not migrate from one processor to the next. It is the only way to present the partitions in a (best) connected manner. The small peaks exceeding 8 processors show the points in time where new jobs are started on the machine and the system is overloaded for a short time.

The presented schedules show a major advantage of the APM: its ability to use the whole system at any time. There are no idle resources like in the optimal offline schedule for the moldable case. The APM using *equiP* shows better response times in this small example than the policy based on accumulated speedup. On the other hand, the end time of the scenario is increased (1380 vs. 1303 seconds). But it is still comparable to the optimal offline schedule (1379 seconds). The model of runtime introduced in section 4 estimates for job C a very good scalability. This leads to the assignment of a dominating portion of processors during its runtime (cf. fig. 2, middle schedule). This strategy lengthens the runtime of jobs estimated to be less efficient but decreases the runtime for the whole scenario. The quality of the schedule shows that an estimation error (here obvious for jobs A vs. C) does not reduce the benefit of the APM too much.

	t_j	runtime [s] (speedup) on p processors			
	[s]	1	2	4	8
A	0	2316 (1.0)	1005 (2.3)	552 (4.2)	340 (6.8)
B	100	1280 (1.0)	748 (1.7)	497 (2.6)	362 (3.5)
C	200	5026 (1.0)	2562 (2.0)	1340 (3.8)	615 (8.2)
D	500	845 (1.0)	629 (1.3)	525 (1.6)	520 (1.6)

Table 1. Description of the presented scenario.

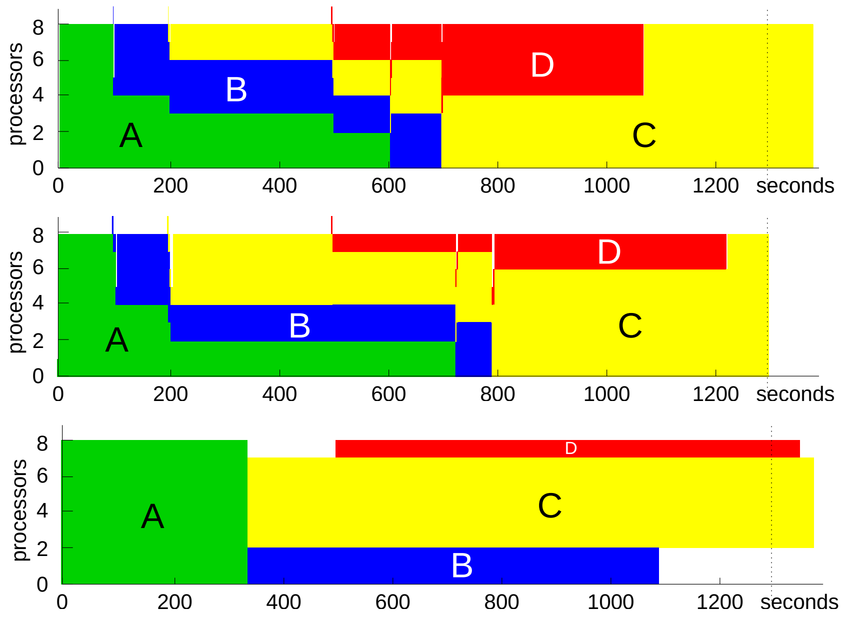

Fig. 2. Different scheduling policies. From top to bottom: equipartition (*equiP*), accumulated speedup (*maxS**, both by APM), and optimal offline schedule (with moldable tasks). The dotted lines at the right end of the graphs represent the end time of the shortest schedule.

The offline schedule shows the problems if only moldability is allowed. Job A scales well, but not good enough to use eight processors efficiently. On the other hand, using less processors increases its runtime and displaces the starting time of job C further into the future. Using one processor leads to a longer runtime, the assignment of two processors would occupy the necessary resources for jobs B and D. Inefficient usage of resources over a longer time have to be accepted in comparison to schedules based on malleable jobs.

The same problem applies if a scheduler for a space-sharing environment uses estimated runtimes and scalability for its calculations. Small prediction errors might lead to different partition sizes and therefore to a less efficient schedule. In our approach temporarily imprecise estimates of the speedup have minor influence on the overall quality of the schedule.

6 Perspectives and Future Work

The extension of the concept to other applications is currently under progress.

At the moment, monitoring of the application is achieved by introducing special function calls in program's main loop. For the setting we have, with access to the source code, this is not a problem. However, it would be nice to find other means for that, e. g. logging facilities of the OS or using processor operation counters [14]. Additionally, the server should consider the system load which is generated by applications not under the server's control.

Currently, this concept lacks fault tolerance in case the central APM crashes or is unreachable. As mentioned above the system load generated by the APM is neglectable low so it can be run on the SMP system. This makes the server independent from network failures. To protect against server crashes a mechanism to restart it after a crash will be sufficient in most situations. Running applications cannot be affected by a disappeared server, and continue their run with the last given resource assignment. They just have to check if the APM is available again and re-connect then.

Each job should inform the server at connection time about the minimum and maximum number of processors it can handle reasonably, i. e. purely sequential jobs tell the server that they can handle not more than one processor. Additionally, a priority mechanism can be integrated easily. Just by introducing a factor to the differential speedup values a weighting of jobs can be done.

Also other metrics are conceivable to maximize besides of maximized accumulated speedup of all applications or utilization of the machine. If applications can provide information about their current state of completion (e. g. already computed steps) the applications' response times can be minimized. However, applications are necessary where the "length" (runtime) is known or can be estimated in advance. A guaranteed end time is also achievable with this information because the APM can then try to assign more processors to an application.

As we developed our concept for shared-memory applications running on SMPs it is adoptable to distributed-memory machines, like clusters, with message-passing. PVM and also the MPI-2 standard both support the dynamic creation and termination of processes during the runtime of an application [11, 17, 9]. A combination with a task migrator can lead to interesting scenarios. One could think of a pool (or a metacomputer) of SMPs and clusters and an APM server. Now applications can migrate from one machine to another (binary compatible) and simultaneously change their degree of parallelism. For that the costs of rebalancing the data after changing the parallelism of an application have to be considered and should be integrated in the model. Thereby an 100 % utilization of all machines is achievable combined with enough flexibility for users who want to start non-malleable jobs locally. In that case malleable and migrateable jobs are either downsized or migrated to another machine.

7 Conclusions

In this paper we have presented an architecture for managing malleable applications on large shared-memory systems efficiently and comfortable. The advantages of time- and space-sharing are combined with respect to the prerequisites of parallel applications. This approach achieves a 100 % system utilisation all the time and guarantees that jobs will start immediately after submission.

Different policies are implemented in the *application parallelism manager* (APM) for achieving different objectives: *maxS** maximises the accumulated speedup of all malleable applications whereas *equiP* equally distributes available resources to. For *maxS** speedup estimations based on an enhancement of Amdahl's Law are used to determine the number of threads to be assigned.

A first implementation of the APM was deployed on a 16-processor SMP to increase the systems usability by achieving higher throughput. Compared to an optimal offline schedule both implemented policies have shown good results. *equiP* shortens the average response time by accepting less efficient system utilisation. *maxS** improves the total runtime and thereby increases the efficiency of the system. Malleable applications managed by the APM have proven to be superior compared to scheduling of moldable jobs.

References

1. S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29(2):459–473, April 2000.
2. G. M. Amdahl. Validity of the single processor approach to achieve large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 31, pages 483–485, 1967.
3. R. Diekmann, J. Hungershöfer, M. Lux, L. Taenzer, and J.-M. Wierum. Efficient contact search for finite element analysis. In *Proc. ECCOMAS*, 2000.
4. D. G. Feitelson. Packing schemes for gang scheduling. *Lecture Notes in Computer Science*, 1162:89–101, 1996.
5. D. G. Feitelson and M. A. Jette. Improved utilization and responsiveness with gang scheduling. *Lecture Notes in Computer Science*, 1291:238–262, 1997.
6. D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. *Lecture Notes in Computer Science*, 1162:1–26, 1996.
7. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and K. C. Sevcik. Theory and practice in parallel job scheduling. *Lecture Notes in Computer Science*, 1291:1–34, 1997.
8. Dror G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Research report rc 19790 (87657), IBM T.J. Watson Research Center, Yorktown Heights, NY, February 1995.
9. Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org.
10. V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *Proc. HiPC*, 2000.
11. K.A. Iskra, Z.W. Hendrikse, G.D. van Albada, B.J. Overeinder, P.M.A. Sloot, and J. Gehring. Experiments with migration of message passing tasks. *Lecture Nores in Computer Science*, 1971:203–213, 2000.
12. T. Kellermeier. Loadbalancing and resource distribution for parallel applications on shared-memory systems (in German). Master's thesis, Paderborn University, 2000.
13. E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. *Lecture Notes in Computer Science*, 949:165–181, 1995.

14. J. Simon, M. Vieth, and R. Weicker. Workload analysis of computation intensive tasks: Case study on SPEC CPU95 benchmarks. In *Proc. Euro-Par*, August 26-29 1997.
15. W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. *Lecture Notes in Computer Science*, 1459:122–142, 1998.
16. W. Smith, I. Foster, and V. Taylor. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proc. of IPPS/SPDP '99 Workshop on Job Scheduling Strategies for Parallel Processing*, 1999.
17. G. D. Van Albada, J. Clinckemaillie, A. H. L. Emmen, and J. Gehring. Dynamite-blasting obstacles to parallel cluster computing. *Lecture Notes in Computer Science*, 1593:300–310, 1999.