Scheduling Divisible Workloads Using the Adaptive Time Factoring Algorithm*

Tiago Ferreto and César De Rose

Catholic University of Rio Grande do Sul (PUCRS),
Faculty of Informatics, Porto Alegre, Brazil
{ferreto, derose}@inf.pucrs.br

Abstract. In the past years a vast amount of work has been done in order to improve the basic scheduling algorithms for master/slave computations. One of the main results from this is that the workload of the tasks may be adapted during the execution, using either a fixed increment or decrement (*e.g.* based on an arithmetical or geometrical ratio) or a more sophisticated function to adapt the workload. Currently, the most efficient solutions are all based on some kind of evaluation of the slaves' capacities done exclusively by the master. We propose in this paper the Adaptive Time Factoring scheduling algorithm, which uses a different approach distributing the scheduling between slaves and master. The master computes, using the Factoring algorithm, a time slice to be used by each slave for processing, and the slave predicts the correct workload size it should receive in order to accomplish this time slice. The prediction is based on a performance model located on each slave which is refined during the execution of the application in order to provide better predictions. We evaluated the proposed algorithm using a synthetic testbed and compared the obtained results with other scheduling algorithms.

1 Introduction

Load balancing has been an ongoing issue for decades. Algorithms based on list-scheduling which manage a list of ready to execute tasks that are sent to slave processors are mainly used because of their suitability to dynamically evolving computations, and also because they cope with heterogeneous resources, since when one processor has finished his work it simply gets more work from the list. This is a simply way to automatic compensate for the differences in the performance of the slaves.

A vast amount of work has been done in order to improve the basic algorithms for master/slave computations. One of the main features concerning load balancing that resulted from this is that the workload of the tasks may be adapted during the execution, using either a fixed increment or decrement (*e.g.* based on an arithmetical or geometrical ratio) or a more sophisticated function to adapt the workload. We present a briefly review of some of these techniques in Section 2.

Yet the solutions presented are all based on some evaluation by the master of the slaves' capacities and of the tasks workload. This implies a significant overhead since the master has to maintain some kind of information about its slaves. We present in

* This research was done in cooperation with HP-Brazil.

M. Hobbs, A. Goscinski, and W. Zhou. (Eds.): ICA3PP 2005, LNCS 3719, pp. 232–239, 2005.

this paper the Adaptive Time Factoring scheduling algorithm, which uses a different approach distributing the scheduling between slaves and master. The master computes, using the Factoring algorithm, a time slice to be used by each slave for processing, and the slave predicts the correct workload size it should receive in order to accomplish this time slice. The prediction is based on a performance model located on each slave which is refined during the execution of the application in order to provide better predictions.

In this paper we review in Section 2 some scheduling algorithms used for master/slave applications with a brief state of the art for each one. Section 3 presents our algorithm and the way each slave can evaluate its capacities. In order to validate our algorithm we devised a synthetic small testbed and Section 4 shows the measurement results that we have obtained using the algorithm proposed in comparison to other algorithms. At last we draw some conclusions about our contribution.

2 Related Work

We present below some classic self-scheduling algorithms proposed in the literature. Self-scheduling [1] represents a large class of dynamic centralized loop scheduling methods. These methods divide the total workload based on a specific distribution, providing a natural load balancing to the application during its execution. We present also some adaptive algorithms that add extensions to the classic self-scheduling algorithms in order to support heterogeneity and adaptability. They consider the load variation in the system environment and adjust the size of the chunks delivered to each processor dynamically. This class of algorithms presents a good performance on dynamic and heterogeneous environments based on its ability to adapt itself to the changes in the environment during the execution of an application.

The Pure Self-scheduling or Workqueue scheduling algorithm divides equally the workload in several chunks. A processor obtains a new chunk whenever it becomes idle. Due to the scheduling overhead and communication latency incurred in each scheduling operation, the overall finishing time may be greater than optimal.

The Fixed-size Chunking scheduling algorithm [2] proposes that each processor receives chunks with size K each time it becomes idle. Although it is hard to determine the best K value in realistic applications due to the high number of dependable variables, the authors give an approximation for an acceptable fixed chunk-size K (using Pth order statistics to model the last P chunks).

The Guided Self-scheduling algorithm [3], schedules large chunks initially, implying reduced communication/scheduling overheads in the beginning, but at the last steps too many small chunks are assigned generating more overhead [1]. Each time a processor requests for more work, the algorithm assigns to it a chunk of size equal to the size of the remaining workload divided by the total number of processors being used for the computation.

Factoring [4] was specifically designed to handle iterations with execution-time variance. With factoring, iterations are scheduled in batches of P equal-sized chunks. The total size of the chunk per batch is a fixed ratio (α) of the remaining workload, *i.e.* $Remaining\_Workload\ /\ \alpha * Number\_Of\_Processors$.

Weighted Factoring Self Scheduling [5] is an improved loop scheduling algorithm addressing load imbalance in a heterogeneous environment. In this algorithm, processors are dynamically assigned decreasing size chunks of iterations in proportion to their processing speeds.

Adaptive Weighted Factoring [6,7] is an adaptive algorithm based on probabilistic analysis, being able to accommodate load imbalances caused by predictable and unpredictable phenomena. In the Adaptive Weighted Factoring, the weight values are adapted after each iteration in the computation. The newly computed weights are not only based on the performance of particular processors during the previous iteration step, but also on their cumulative performance during all the previous iterations.

Adaptive Factoring [8,9] allows a relaxation of some of the theoretical assumptions imposed by models used in earlier methods, therefore making this technique more robust to any load variations present in the environment and improving the performance of applications characterized by highly irregular behavior. In this algorithm, the weights are dynamically assigned to processors at run time by closely following the rate of change in processor speed. The model used for this method allows the dynamic computation of new weights for each processor, when a new chunk is allocated.

In all algorithms shown above, the information needed to evaluate the best processor to run the remaining workload is centralized at the master process, which is responsible for the decision regarding increasing or decreasing the chunk that is executed by each slave process. We propose in the following Section another approach, where the evaluation of the chunk size to be assigned to each slave is done by the slave itself.

3 Adaptive Time Factoring Scheduling Algorithm

The Adaptive Time Factoring (ATF) scheduling algorithm is, like others algorithms (e.g. Weighted Factoring [5], Adaptive Weighted Factoring [6,7]), based on the decreasing scheme proposed by Factoring [4]. However, instead of decreasing, for each round, the number of tasks to be processed by each slave, it decreases the time slice that each slave should use. Each slave predicts the best chunk size it should process based on the time slice given using a performance model.

The main features of the algorithm are the utilization of time instead of chunk sizes as a scheduling metric, and the distribution of the performance model structure between the slaves. The utilization of time instead of chunk sizes facilitates handling heterogeneous slaves due to better scheduling abstraction, *i.e.* the scheduler can assure that scheduling the same amount of time for different slaves will result in approximately the same completion time benefiting overall performance. The adoption of a distributed performance model managed by the slaves instead of a centralized one at the serevr provides better scalability support avoiding a centralized bottleneck and faster adaptation of the model due to performance variable fluctuations. Based on this data distribution, the algorithm scheduling decision is also distributed between master and slaves. In contrast to other algorithms, the slave also participates at the scheduling decision calculating the chunk size to be processed using its local performance model.

The main goal of the algorithm is to minimize execution time of applications in heterogeneous and dynamic environments. It addresses particularly applications using the master/slave model containing divisible workloads, *i.e.* the total amount of work to be processed can be divided in equal-size chunks.

During the execution of the algorithm, each slave builds an internal performance model which contains the slave's execution and communication time demands to process chunks of the application workload. It enables the prediction of the chunk size to be processed by the slave in order to fully use the time slice given by the master. Detailed information related to the performance model and the prediction method used in the algorithm is presented in Section 3.1.

The Adaptive Time Factoring scheduling algorithm is based on a distributed scheduling method. Each slave computes a chunk sent by the master and, based on its internal performance model, predicts the best chunk size to be computed at the next iteration considering the time slice given by the master. The master distributes time slices in decreasing chunks between the slaves. The decreasing method used is based on the Factoring [4] algorithm with a fixed value $\alpha = 2$.

In order to obtain efficient slave predictions for scheduling, it's necessary to build and refine the performance model in each slave before using the predictions. Due to this requirement, the Adaptive Time Factoring scheduling algorithm is divided in two distinct phases: setup phase and adaptive phase.

The setup phase is used to build the local performance model on each slave and to refine it in order to produce predictions with minimum error. It's like an initial benchmarking of each slave using the application workload. At the beginning, the master sends to each slave a chunk with minimum size and waits for the results. After receiving the results from a slave, it sends another chunk to the slave duplicating its size by a factor of two. It continues this process until it receives a signal from the slave indicating to start with the adaptive phase. This signal is sent when the slave already has an efficient performance model capable of producing good predictions, *i.e.* the model provides minimum error comparing predicted and measured execution times.

The adaptive phase turns over an increasing size mechanism to a decreasing one. However, instead of decreasing the chunk size, it decreases the time slice used by each slave to predict the more appropriate chunk size to be processed. Since the beginning of the algorithm, each time the master sends a chunk to each slave, it includes in the message a time slice for the next round. This time slice is used by the slave to predict the chunk size that it can be execute at the next round. The slave returns a message with the results of the chunk processing, the execution time that it took to process the chunk and the chunk size predicted. This chunk size is only considered at the algorithm after the slave sends the signal to the master in order to start the adaptive phase. The time slice for the round is computed as:

$$timeSlice_{i+1}^{root} = (workload_{i+1} * avgExecTime)/\alpha * nSlaves$$

where $timeSlice_{i+1}^{root}$ is the time slice computed for the next round $(i + 1)$, $workload_{i+1}$ is an estimation of the remaining workload at the next round $(i + 1)$, $avgExecTime$ is the average execution time for a chunk with minimum size, $alpha$ is a parameter of the Factoring algorithm which is fixed to 2 and $nSlaves$ is the total

number of slaves. The average execution time is computed each time the master receives a new result from some slave using chunk size and execution time values. Since the time slice sent is related to the next round, it's necessary to use an estimation of the remaining workload at the next round. It is compute as:

$$workload_{i+1} = workload_i - \frac{nSlaves * timeSlice_i}{avgExecTime}$$

The estimation is based on the subtraction, on the current workload, of the average chunk size that can be processed by all slaves during the current time slice. In order to minimize the gap between slaves completion times, the following rule is used: the first slave in a round computes its time slice as presented above (root time slice), and all the others compute their time slices as:

$$timeSlice_{i+1} = timeSlice_{i+1}^{root} - \delta^j$$

where $timeSlice_{i+1}^{root}$ is the first time slice computed at the beginning of the round, and δ^j is the time taken to set this new time slice since the first time slice of this round has been computed.

The master algorithm for the adaptive phase is presented in Figure 1. It keeps in a loop sending chunks and receiving the results to available slaves until the workload is empty. Before sending the chunk to a slave, it computes the time slice, which depends if the slave is the first to compute this value or not, as described before. It assigns the new chunk size with the slave's prediction size previously returned with the last result, and sends to the slave the chunk and time slice.

At reception, the master receives the result of chunk processing, execution time took to process the chunk and the predicted chunk size for the next round. The average execution time is computed using chunk size and execution time parameters.

Algorithm 1. Adaptive Time Factoring algorithm

```
1:  while workload is not empty do
2:    for each available slave do
3:      if beginning of round i then
4:        compute timeSlice_{i+1}^{root}
5:      else
6:        compute timeSlice_{i+1}
7:      end if
8:      chunk ⇐ predictedSize
9:      send chunk and timeSlice_{i+1}
10:   end for
11:   receive result, execTime and predictedSize
12:   compute avgExecTime
13: end while
```

Due to the existence of a performance model on each slave, any change in the machines (*e.g.* machine turned down, start of a concurrent application) results in an adaptation of the best chunk size to be processed by the slave. It is important to emphasize

that different slaves can be in distinct phases of the algorithm at the same time, *i.e.* some of the machines can be executing at the setup phase and others at the adaptive phase. This condition happens when more slaves are included during the execution.

3.1 Local Prediction of the Computational Load

In order to estimate the most suited workload, a slave needs a performance model for the execution of chunks of size $chunkSize_i$. The model may include various data such as the execution time, memory utilization, etc, used to process a given chunk. In this preliminary version of our prototype we only take into account the execution time.

Given some N values $chunkSize_1, chunkSize_2, \ldots chunkSize_n$ and the slaves's data t (*e.g.* the execution time) the slave has to estimate $t(chunkSize)$. In a multi-parameter model we could use algorithms such as the Singular Value Decomposition [10], one of the most robust for data modeling. It would fit the function t as a linear combination of standard base functions (*e.g.* $x \rightarrow e^x, \sqrt{}$, polynomials, ...).

Nevertheless in the case where t only depends on the processor's speed, an affine model of the time required *vs.* the chunk size to run is most realistic and used by other algorithms [11]. The modeling problem is therefore a basic linear interpolation problem of the measured running time $t_j, j = 1 \ldots n$ *vs.* the chunk size $chunkSize_j$. Besides the estimated coefficients a, b of the affine approximation $t = a + b \times chunkSize$, the correlation coefficient is used to determine the correction of the interpolation and thus decide if a larger chunk should be sent in the initial phase.

The interpolation algorithm is very fast and thus does not prejudice the execution of the application. Moreover, it is trivial for a slave to determine the adapted chunk size, given the execution time t it has to run and the affine model (a, b). Note that in the case of a more complex, non-linear model, it would have to use a more time-consuming algorithm such as a gradient or dichotomic search to solve the $t = f(chunkSize)$ equation.

4 Evaluation

In order to evaluate the performance of the Adaptive Time Factoring scheduling algorithm (ATF) we devised a simple master/slave application and executed it in a heterogeneous cluster. This application consists in w multiplications of two matrices of size $n \times n$. The minimum chunk size is the multiplication of two matrices ($w = 1$). With this application we are able to easily vary the size and the number of chunks to be processed, generating different conditions to evaluate the behavior of our algorithm.

We executed the application in a cluster with 16 machines connected through a Fast-Ethernet network. The testbed consists of four different types of nodes divided in classes, from A to D (4 nodes per machine class). To give an idea of the performance of each machine class Table 1 presents their execution times for the computation of a chunk for three different matrix sizes.

We compared our Adaptive Time Factoring scheduling algorithm (ATF) to the classical Workqueue algorithm (WQ), and to two factoring algorithms, the non-adaptive Factoring Algorithm (FAC), and the Adaptive Weighted Factoring (AWF) with $\alpha = 2$.

Table 1. Execution time for one chunk (one matrix multiplication - $w = 1$)

n	execution time (seconds)			
	Class A	**Class B**	**Class C**	**Class D**
300	0.80	1.00	1.34	1.44
500	5.18	6.89	9.35	9.52
700	14.82	19.74	27.03	27.22

Table 2. Comparison of the execution times of the algorithms (in seconds)

t	n											
	300				**500**				**700**			
	WQ	FAC	AWF	ATF	WQ	FAC	AWF	ATF	WQ	FAC	AWF	ATF
1000	61.74	62.86	61.7	62.15	486.22	483.98	483.94	483.04	1391.57	1385.12	1378.60	1380.34
1500	94.45	93.79	91.38	92.80	727.03	726.86	724.55	725.84	2081.26	2080.94	2068.97	2072.44
2000	122.69	124.63	121.54	124.77	964.97	965.68	963.32	964.99	2767.57	2769.48	2754.77	2756.48
2500	152.76	155.70	158.16	155.05	1208.22	1207.80	1205.15	1206.89	3456.94	3452.83	3438.00	3444.37
3000	190.16	187.5	183.98	181.9	1449.34	1448.31	1442.63	1445.86	4151.01	4144.19	4126.22	4134.71

We used three different matrix sizes (n): 300, 500 and 700, and five number of multiplications for the workloads w: 1000, 1500, 2000, 2500 and 3000. The obtained results are presented in Table 2.

In most cases ATF outperforms WQ and FAC, particularly in bigger matrices. This is expected because of the heterogeneity of the testbed. Adaptive algorithms can adapt the number of chunks scheduled to a node depending on their performance thus making better use of the resources.

ATF has similar results to AWF in all cases (difference around 1% in execution times). We think this is a very promising result considering that AWF is one of the latest algorithms introduced and is also known for having the best results for the kind of measurements we are performing ([6,7]). Besides, we believe that the benefits of adaptability and distributed scheduling presented in ATF, in order to improve scalability and performance, couldn't be explored in our experiments, since the measurements have been done in a small heterogeneous cluster. We believe that using more machines, ATF will eventually overcome AWF.

5 Conclusions

In this paper we presented the Adaptive Time Factoring (ATF) scheduling algorithm. It is based on a distributed scheduling method, in which each slave computes a chunk sent by the master and, based on its internal performance model, predicts the best chunk size to be computed at the next iteration considering the time slice given by the master. The master distributes time slices between the slaves using a decreasing method based on the Factoring algorithm.

We presented experimental measurements with ATF in a heterogeneous platform and compared it to other algorithms. The results show that ATF outperforms Workqueue

and Factoring in most cases, particularly in bigger matrices. ATF showed also similar results to Adaptive Weighted Factoring in all cases (difference around 1% in execution times). We think this is a very promising result considering that AWF is known for having the best results for the kind of measurements we performed. We also believe that ATF, due to its distributed scheduling mechanism, will eventually overcome AWF in a testbed with more machines.

References

1. Chronopoulos, A.T., Andoine, R., Benche, M., Grosu, D.: A class of loop self-scheduling for heterogeneous clusters. In: Proceedings of CLUSTER'2001. (2001)
2. Kruskal, C.P., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Transactions on Software Engineering **11** (1985) 1001 – 1016
3. Polychronopoulos, C.D., Kuck, D.J.: Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Transactions on Computers **36** (1987) 1425–1439
4. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: A method for scheduling parallel loops. Communications of the ACM **35** (1992) 90–101
5. Hummel, S.F., Schmidt, J.P., Uma, R.N., Wein, J.: Load-sharing in heterogeneous systems via weighted factoring. In: Proceedings of the 8th Symposium on Parallel Algorithms and Architectures. (1997)
6. Banicescu, I., Velusamy, V.: Performance of scheduling scientific applications with adaptive weighted factoring. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001) - Heterogeneous Computing Workshop, San Francisco (2001)
7. Banicescu, I., amd Johnny Devaprasad, V.V.: On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Cluster Computing: The Journal of Networks, Software Tools and Applications **6** (2003) 213–226
8. Banicescu, I., Liu, Z.: Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. In: Proceedings of the High Performance Computing Symposium (HPC 2000)ac, Washington (2000) 122–129
9. Banicescu, I., Velusamy, V.: Load balancing highly irregular computations with the adaptive factoring. In: Proceedings of the IEEE - International Parallel and Distributed Processing Symposium (IPDPS 2002) - Heterogeneous Computing Workshop, Fort Lauderdale (2002)
10. Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: Numerical Recipes in C: The Art of Scientific Computing. 2nd edn. Cambridge University Press (1993)
11. Beaumont, O., Legrand, A., Robert, Y.: Scheduling divisible workloads on heterogeneous platforms. Parallel Computing **29** (2003) 1121–1152