Dynamic Pipeline Mapping (DPM)*

A. Moreno[1], E. César[2], A. Guevara[2], J. Sorribes[2],
T. Margalef[2], and E. Luque[2]

[1] Escola Universitària Salesiana de Sarrià, Passeig Sant Joan Bosco 74, 08017 Barcelona, Spain
amoreno@euss.es

[2] Departament Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
{eduardo.cesar@,alex.guevara@caos.,joan.sorribes@,
tomas.margalef@,emilio.luque@}uab.es

Abstract. Parallel/distributed application development is an extremely difficult task for non-expert programmers, and support tools are therefore needed for all phases of the development cycle of this kind of applications. In particular, dynamic performance tuning tools can take advantage of the knowledge about the application's structure given by a skeleton based programming tool. This study shows the definition of a strategy for dynamically improving the performance of pipeline applications. This strategy, which has been called Dynamic Pipeline Mapping, improves the application's throughput by gathering the pipe's fastest stages and replicating its slowest ones. We have evaluated the new algorithm by experimentation and simulation, and results show that DPM leads to significant performance improvements.

1 Introduction

Parallel/distributed programming constitutes a highly promising approach to the improvement of the performance of many applications. However, in comparison to sequential programming, several new problems have emerged in all phases of the development cycle. One of the best ways to solve these problems would be to develop tools that support the design, coding, and analysis and/or tuning of parallel/distributed applications.

In the particular case of performance analysis and/or tuning, it is important to note that the best way for analyzing and tuning parallel/distributed applications depends on some of their behavioral characteristics. If the application being tuned behaves in a regular way, then a static analysis will be sufficient. However, if the application changes its behavior from execution to execution, or even within a single execution, then dynamic monitoring and tuning techniques should be used instead.

The key issue in dynamic monitoring and tuning is that decisions must be taken efficiently while minimizing intrusion on the application. We show that this

* This work was supported by MEC under contract TIN2007-64974.

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 295–304, 2008.
© Springer-Verlag Berlin Heidelberg 2008

is easier to achieve when the tuning tool uses a performance model associated to the structure of the application. Knowing the application's structure is not a problem if a programming tool, based on the use of skeletons or frameworks, is used for its development.

This study, focused on a comprehensive performance model associated to the Pipeline framework, represents a further step to our previous contributions [1], [2] and [3] to the development of performance models associated to the application structure for dynamic performance tuning.

The Pipeline framework is a well-known parallel programming structure used as the most direct way to implement algorithms that consist of performing an orderly sequence of essentially identical calculations on a sequence of inputs. Each of these calculations can be broken down into a certain number of different stages, which can be concurrently applied to different inputs.

The possible inefficiencies of pipelined applications are also well known. At first, the concurrency is limited at the beginning of the computation as the pipe is filled, and at the end of the computation as the pipe is drained. Programmers should deal with this inefficiency at the design phase of the application because the way to avoid it is to assure that the number of calculations the application will perform are substantially higher than the number of stages of the pipe.

Secondly, it is important not to have any significant differences among the computational efforts of the pipe stages, because the application throughput of a pipe is determined by its slowest stage. This is the most important inefficiency of this structure, and the most difficult to overcome because it does not depend exclusively on the application design, but also on run-time conditions. Consequently, our study is focused on dealing with this drawback, which is suitable for being solved dynamically.

This paper is organized in six sections, the first is this introduction, the second reviews some relevant related studies, the third shows the expressions defined in order to model the stages of a pipeline, the fourth describes the Dynamic Pipeline Mapping algorithm we propose in this study, the fifth assesses the algorithm through simulation and synthetic experiments, and finally, the sixth section shows the main conclusions of this study.

2 Related Work

There are several studies about the performance modelling of pipeline applications. Some of them propose highly constrained models, such as Subhlok and Vondram [4] and Lee et al. [5]. The former proposes a mapping algorithm which optimizes latency under some throughput constraints for purely linear pipes, while the latter defines a policy for maximizing the throughput of homogeneous pipelines (those where each stage processes a homogeneous group of tasks).

Nevertheless, there are studies, such as Hoang and Rabey [6], focused on more generic pipeline applications. This work proposes an algorithm for maximizing the throughput that has been improved lately by Yang et al. [7] and Guirado et al. [8] considering that the application performs several iterations.

On the other hand, Almeida et al. [9] have defined an algorithm, based on gathering stages on a processor, to improve the performance of a pipeline application on a cluster of heterogeneous processors. Their model comprehends the whole life of the application, including the filling-in and draining phases of the execution of a pipeline, which they try to minimize.

Another highly relevant study in this area is the Murray Cole's group of the School of Informatics of the University of Edinburgh. They have realized that using skeletons carries with it considerable information about implied scheduling dependences and have decided to use process algebras (specifically PEPA [10]) for modeling them.

All these studies are mainly focused on the static analysis of the pipeline application. This is the main difference regarding our model proposal, which is intended to dynamically tuning the application's performance. Consequently, we consider that filling-in and draining the pipe are transient phases, whose inefficiencies cannot be solved dynamically. On the other hand, we improve the application's performance by gathering fast consecutive stages in the same processor, which leads to lower communication costs, and by replicating the slowest stages in several processors in order to increase their throughput.

3 Modelling the Stages of a Pipeline Application

This section introduces the mathematical expressions defined to model the performance of a replicated stage, as well as those defined to model the performance of a set of stages assigned to the same processor. For these expressions we will use the following terminology:

- P = Number of available processors.
- N = Number of pipeline stages.
- Tr_i = Inverse throughput of stage i, which is the time needed by the stage i to process each task assuming that this stage is not sharing processor with any other. We call this the *independent production time* of stage i.
- Tr_i' = Inverse throughput of stage i when it is sharing processor with other stages. We call this the *grouped production time* of stage i.
- $Tr_{ireplica}$ = Inverse throughput of stage i when it has been replicated in several processors. We call this the *replicated production time* of stage i.
- Tp_k = Inverse throughput of processor k, which is the elapsed time from moment a task arrives to the first stage until it leaves the last stage assigned to processor k. We call this the *production time of processor* k.

3.1 Replicated Stage Performance Model

It has been earlier shown [8] that the overall throughput of a pipeline application is determined by slowest stage. Consequently, the performance of the application can be improved by replicating that particular stage. We have chosen a very simple approach for modeling a replicated stage, which consists of dividing

the independent production time of the non replicated stage by the number of processors used to replicate it (P_i):

$$Tr_{ireplica} = \frac{Tr_i}{P_i}. \tag{1}$$

This is an oversimplification of the real system because it does not take into consideration the overhead introduced by the extra communications, nor the time needed to manage the replicas. However, it is good enough for our purposes and helps keeping the overall performance model simple.

3.2 Grouped Stages Performance Model

Assembling several fast consecutive stages in the same processor leads to performance improvements because it decreases the communication cost and, even more important, it releases processing resources that can be dedicated to the replication of the slowest stages.

We have modeled a set of grouped stages as the production time of the processor there are assigned to (Tp_k). It is calculated adding up the independent production times (Tr_i) of each gathered stage:

$$Tp_k = \sum Tr_i. \tag{2}$$

We will need the independent production time (Tr_i) of an individual stage in a group in order to evaluate the overall pipeline performance model. However, when a stage is sharing processor with others stages, it is not possible to measure its Tr_i because, as far as we are only inserting instrumentation in the application code, any measure will include the stage execution time plus the time it has spent waiting for the processor while other stages were executing. This is the reason why we have given a different name (Tr'_i) to the grouped production time of a gathered stage. Consequently, we will use this time (Tr'_i) and the total processor production time (Tp_k) to estimate the independent production time of a stage in a group according to Eq. (3), which assigns to a stage a share of the overall group execution time (Tp_k) proportional to its observed weigh ($Tr'_i / \sum Tr'_i$) within the group.

$$Tr_i = Tp_k \frac{Tr'_i}{\sum Tr'_i}. \tag{3}$$

4 Dynamic Pipeline Mapping

A pipeline application will reach its best performance when the processor production times (Tp_k) of every processor holding one or more stages, and the production time of every replicated stage ($Tr_{ireplica}$) are the same, it is, when the workload is balanced among the available processors. Moreover, in any other case the slowest stage will determine the throughput of the application.

We have defined an algorithm called Dynamic Pipeline Mapping with the objective of improving the performance of a pipeline application. The main idea is to gather fast consecutive stages and to replicate slower ones in order to find the best possible mapping leading to the best possible load balancing.

DPM consists of three phases, which are executed iteratively at runtime in order to adapt the mapping to the current conditions of the application:

1. *Measurement or estimation of independent production time (Tr_i).* It is obtained for every pipe stage, including those assembled with other stages in the same processor. In this case Eq. (3) will be used to estimate this value.
2. *Alternative mapping proposal.* Knowing the (Tr_i) of every stage makes it possible to calculate the ideal throughput of the pipe ($\frac{P}{\sum Tr_i}$) and, given the current mapping of the application, it is possible to propose a new distribution of stages that gets as closer as possible to this throughput. We use the term *mean production time*, Tp_{ideal}, as the inverse of the ideal throughput of the pipe.
3. *Proposal evaluation and implementation.* The proposal generated in the second phase is evaluated comparing its predicted throughput with the current one. If the gain is good enough then it will be worth implementing the new mapping.

From the point of view of this study, the second phase is the core of this algorithm. It consists of two steps, in the first one, aimed to release resources, the whole pipe is traversed from its first stage identifying groups of stages whose production times are below the mean, while in the second step the freed resources are used for increasing the throughput of the slowest stages.

In this first step, the pipe is traversed from its first stage. If a replicated, a single or a set of grouped stages are found whose production time is below the mean production time (Tp_{ideal}), some resources will be released according to the following rules:

– *Replicated stage.* The desirable number of replicas (P_i) is calculated using Eq. (4). Then the remaining replicas are proposed for elimination, releasing the processors they were assigned to.

$$P_i = \frac{Tr_i}{Tp_{ideal}} \tag{4}$$

– *Set of single or grouped stages.* Using Eq. (5) a new gathering, including more stages, is proposed in order to release some processors.

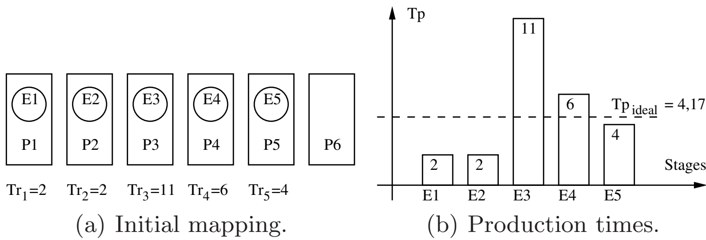
$$Tp_{ideal} = \sum Tr_i \tag{5}$$

On the other hand, if the algorithm founds a replicated, single, or a set of grouped stages whose production time is above the mean production time (Tp_{ideal}), it will use some spare processors in order to approach its production time to Tp_{ideal} in the following way:

- *Replicated or single stage.* The desirable number of replicas (P_i) is calculated using Eq. (4) and the needed replicas must be added. However, this process will be performed at the next step of the algorithm when the number of available resources are known.
- *Grouped stages.* The current gathering should be broken up in order to improve its production time. We use Eq. (5) to determine the partition of the gathering.

Finally, in the second step, knowing the available resources and the stages or replicated stages that need processors, the algorithm assigns processors to stages or replicated stages that need them, because these are the ones which have greatest influence on the global performance of the pipeline application.

In order to clarify how this algorithm would work, we propose a particular application example. It is assumed that initially the application state is the one shown in Figure 1(a). There are 5 stages (E1, E2 ... E5) and 6 processors (P1, P2 ... P6). Tr_i is the independent production time of stage i in time units.



(a) Initial mapping. (b) Production times.

Fig. 1. Application example, initial mapping of 5 stages and 6 processors

First, the mean production time, Tp_{ideal}, is calculated:

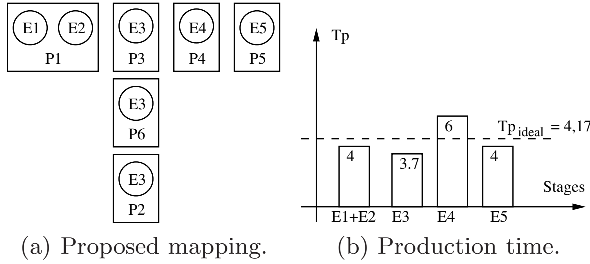
$$Tp_{ideal} = \frac{\sum Tr_i}{P} = 4.17. \tag{6}$$

The relationship between the mean production time and the independent production of each stage can be seen in Figure 1(b).

Then, the second phase of the algorithm is executed. First we evaluate the Eq. (5) and we see that if we group the stages E1 and E2, the gathering production time will be 2+2=4, a value closer to the ideal production time 4.17. As a consequence processor P2 would be released to be used later.

Next, the stage E3 is above the mean production time stage and its replication with the free resources is proposed. The desirable number of replicas (P_i) is calculated using Eq. (7) and we obtain 2.64 as result. Finally, we round up to 3 processors. Figure 2(a) shows the proposed mapping and Figure 1(b) the respective production times.

$$P_i = \frac{Tr_i}{Tp_{ideal}} = \frac{11}{4.17} = 2.64 \tag{7}$$



(a) Proposed mapping. (b) Production time.

Fig. 2. Application example, mapping proposed by Dynamic Pipeline Mapping algorithm

In the third phase of the algorithm the proposal is evaluated. As the pipeline production time is defined by the maximum production time, with the proposed mapping this value changes from 11 (stage E3 in Figure 1(a)) to 6 (stage E4 in Figure 2(a)). Consequently, the throughput of the pipeline would be improved significantly.

5 Algorithm Assessment

This section describes the evaluation of the Dynamic Pipeline Mapping algorithm through experimentation and simulation. We have used simulation because it allows the systematic evaluation of hundreds of cases with few resources and in a reasonable period of time, and we have validated simulation results executing several scenarios in a cluster of workstations.

The simulation method considers that the independent production time (Tr_i) of each stage can be modeled as a random variable with a normal distribution and with parametric mean (μ) and standard deviation (σ). Therefore, in order to build a particular execution scenario, a sample value of each random variable is generated. Next, the execution time of the application is calculated for a MPI-like mapping (Te_{MPI}) and, finally, the DPM algorithm is applied and the execution time (Te_{DPM}) for its proposed mapping is also calculated.

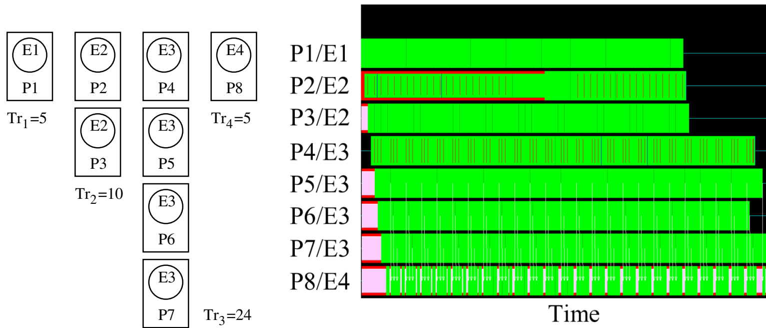
The simulation examples are shown in Table 1. Independent production time (Tr_i) samples were obtained from a normal distributed random variable with $\mu = 10$ and $\sigma = 8$ time units. Table 1 shows the ratio between execution time

Table 1. Simulation results of 1000 executions scenarios for 32 processors pipeline

Stages	Mean of Te_{MPI}/Te_{DPM}	Standard deviation of Te_{MPI}/Te_{DPM}
16	1.36	0.23
32	1.55	0.26
64	1.24	0.33

of MPI-like mapping (Te_{MPI}) and execution time of DPM algorithm mapping (Te_{MPI}) of 1000 scenarios. Results clearly indicate that for an overwhelming number of cases applying DPM leads to significant improvements in the application performance. Although, the algorithm obviously produces better results when there are more available processors than stages.

In order to validate simulation results we have written a MPI synthetic pipeline program that allows the definition of the number of stages of the pipe, the number and size of data elements to be processed, the computation time associated to each stage, the number of instances of each replicated stage, and the stages included in each grouped stage.



(a) Mapping of 4 stages and 8 processors case.

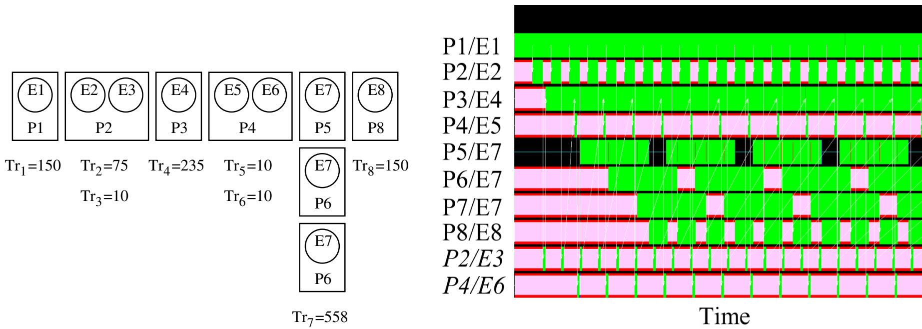
(b) Execution trace with DPM algorithm.

Fig. 3. First experimental scenario: 4-stage pipe on a cluster of 8 homogeneous workstations (independent production time,Tr, is in ms)

Figure 3 shows the execution trace of a 4-stage pipe on a cluster of 8 homogeneous workstations. The number of data elements processed are 100 and the message size is 100 bytes (low communication cost). In addition, the computation load is 5 ms for stages 1 and 4, 10 ms for stage 2, and 24 ms for stage 4.

We have compared the execution time of the MPI-like mapping against the DPM proposed mapping. In this case DPM has several available processors that can be used for replicating slower stages. Consequently, the algorithm proposes the mapping shown in Figure 3(a), where one extra instance of stage 2 and three new instances of stage 4 are created. The execution time of the application using the MPI-like mapping is 2.43 s, while the one for the DPM proposed mapping is 0.64 s, which is 3.8 times better.

Figure 4 shows the results for a less favorable case, where an 8-stage pipe is executed on the same cluster using the MPI-like mapping and the DPM proposed mapping (Figure 4(a)). The number and size of messages are the same than in the previous example. However, in this case the computation load is 150 ms for



(a) Mapping of 8 stages and 8 processors case.

(b) Execution trace with DPM algorithm.

Fig. 4. Second experimental scenario: 8-stage pipe on a cluster of 8 homogeneous workstations (independent production time,Tr, is in ms)

stages 1 and 8; 75 ms for stage 2; 10 ms for stages 3, 5, and 6; 235 ms for stage 4; and 558 ms for stage 7.

In this case DPM has no spare processors at the beginning, so it proposes to group stages 1 and 2 (grouped production time of 85 ms), and stages 4 and 5 (grouped production time of 20 ms). These actions liberate 2 processors that are used to introduce 2 extra instances of stage 7, leading to the mapping shown in Figure 4(a). The execution time of the application using the MPI-like mapping is 62.03 s, while the one for the DPM proposed mapping is 26.6 s, which is 3.33 times better. These results show that for both cases the DPM proposed mapping led to significant performance gains.

6 Conclusions

We have proposed a strategy for dynamically improving the performance of pipeline applications. The resulting algorithm, which has been called Dynamic Pipeline Mapping, is intended to improve the application's throughput by gathering the pipe's fastest stages and replicating its slowest ones, and it is based on simple analytical models of the pipeline stages behavior.

In addition, we have shown simulated and experimental results that demonstrate that the DPM proposed mappings usually lead to significant performance improvements that justify the effort of dynamically implementing the necessary changes.

Completing the analytical models to include efficient communication management considerations is the next challenge. However, we believe that DPM is a relevant contribution for developing model based dynamic performance tuning tools.

References

1. Cesar, E., Moreno, A., Sorribes, J., Luque, E.: Modeling master/worker applications for automatic performance tuning. Parallel Comput. 32(7), 568–589 (2006)
2. Cesar, E., Sorribes, J., Luque, E.: Modeling pipeline applications in poetries. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 83–95. Springer, Heidelberg (2005)
3. Morajko, A., Cesar, E., Caymes-Scutari, P., Margalef, T., Sorribes, J., Luque, E.: Automatic tuning of master/worker applications. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 95–103. Springer, Heidelberg (2005)
4. Subhlok, J., Vondran, G.: Optimal use of mixed task and data parallelism for pipelined computations. Journal of Parallel and Distributed Computing 60(3), 297–319 (2000)
5. Lee, M., Liu, W., Prasanna, V.K.: A mapping methodology for designing software task pipelines for embedded signal processing. In: IPPS/SPDP Workshops, pp. 937–944 (1998)
6. Hoang, P.D., Rabaey, J.: Scheduling of dsp programs onto multiprocessors for maximum throughput. IEEE Transactions on Signal Processing 41(6), 2225–2235 (1993)
7. Yang, M.-T., Kasturi, R., Sivasubramaniam, A.: A pipeline-based approach for scheduling video processing algorithms on now. Transactions on Parallel and Distributed Systems 14(2), 119–130 (2003)
8. Guirado, F., Ripoll, A., Roig, C., Luque, E.: Exploitation of parallelism for applications with an input data stream: optimal resource-throughput tradeoffs. In: PDP 2005. 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 9-11 February, pp. 170–178 (2005)
9. Almeida, F., González, D., Moreno, L.M., Rodríguez, C.: An analytical model for pipeline algorithms on heterogeneous clusters. In: Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, London, UK, pp. 441–449. Springer, Heidelberg (2002)
10. Gilmore, S., Hillston, J., Kloul, L., Ribaudo, M.: Pepa nets: a structured performance modelling formalism. Perform. Eval. 54(2), 79–104 (2003)