Evaluating the Performance of Skeleton-Based High Level Parallel Programs

Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston

School of Informatics, The University of Edinburgh, James Clerk Maxwell Building,
The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
enhancers@inf.ed.ac.uk,
http://homepages.inf.ed.ac.uk/stg/research/ENHANCE/

Abstract. We show in this paper how to evaluate the performance of skeleton-based high level parallel programs. Since many applications follow some commonly used algorithmic skeletons, we identify such skeletons and model them with process algebra in order to get relevant information about the performance of the application, and be able to take some "good" scheduling decisions. This concept is illustrated through the case study of the Pipeline skeleton, and a tool which generates automatically a set of models and solves them is presented. Some numerical results are provided, proving the efficiency of this approach.

1 Introduction

One of the most promising technical innovations in present-day computing is the invention of Grid technologies which harness the computational power of widely distributed collections of computers [6]. Designing an application for the Grid raises difficult issues of resource allocation and scheduling (roughly speaking, how to decide which computer does what, and when, and how they interact). These issues are made all the more complex by the inherent unpredictability of resource availability and performance. For example, a supercomputer may be required for a more important task, or the Internet connections required by the application may be particularly busy.

In this context of Grid programming, skeleton based programming [3,11,5] recognizes that many real applications draw from a range of well known solution paradigms and seeks to make it easy for an application developer to tailor such a paradigm to a specific problem. In this high-level approach to parallel programming, powerful structuring concepts are presented to the application programmer as a library of pre-defined 'skeletons'. As with other high-level programming models the emphasis is on providing generic polymorphic routines which structure programs in clearly-delineated ways. Skeletal parallel programming supports reasoning about parallel programs in order to remove programming errors. It enhances modularity and configurability in order to aid modification, porting and maintenance activities. In the present work we focus on the Edinburgh Skeleton Library (eSkel) [4]. eSkel is an MPI-based library which has been designed for

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 289–296, 2004.

SMP and cluster computing and is now being considered for Grid applications using Grid-enabled versions of MPI such as MPICH-G2 [10].

The use of a particular skeleton carries with it considerable information about implied scheduling dependencies. By modelling these with stochastic process algebras such as PEPA [9], and thereby being able to include aspects of uncertainty which are inherent to Grid computing, we believe that we will be able to underpin systems which can make better scheduling decisions than less sophisticated approaches. Most significantly, since this modelling process can be automated, and since Grid technology provides facilities for dynamic monitoring of resource performance, our approach will support *adaptive* rescheduling of applications.

Some related projects obtain performance information from the Grid with benchmarking and monitoring techniques [2,12]. In the ICENI project [7], performance models are used to improve the scheduling decisions, but these are just graphs which approximate data obtained experimentally. Moreover, there is no upper-level layer based on skeletons in any of these approaches.

Other recent work considers the use of skeleton programs within grid nodes to improve the quality of cost information [1]. Each server provides a simple function capturing the cost of its implementation of each skeleton. In an application, each skeleton therefore runs only on one server, and the goal of scheduling is to select the most appropriate such servers within the wider context of the application and supporting grid. In contrast, our approach considers single skeletons which span the grid. Moreover, we use modelling techniques to estimate performance.

Our main contribution is based on the idea of using performance models to enhance the performance of grid applications. We propose to model skeletons in a generic way to obtain significant performance results which may be used to reschedule the application dynamically. To the best of our knowledge, this kind of work has never been done before. We show in this paper how we can obtain significant results on a first case study based on the pipeline skeleton.

In the next section, we present the pipeline and a model of the skeleton. Then we explain how to solve the model with the PEPA workbench in order to get relevant information (Section 3). In section 4 we present a tool which automatically determines the best mapping to use for the application, by first generating a set of models, then solving them and comparing the results. Some numerical results on the pipeline application are provided. Finally we give some conclusions.

2 The Pipeline Skeleton

Many parallel algorithms can be characterized and classified by their adherence to one or more of a number of generic algorithmic skeletons [11,3,5]. We focus in this paper on the concept of pipeline parallelism, which is of well-proven usefulness in several applications.

We recall briefly the principle of the pipeline skeleton, and then we explain how we can model it with a process algebra.

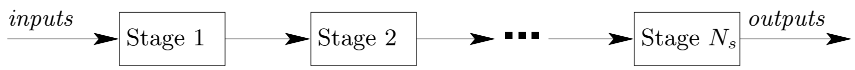

Fig. 1. The pipeline application

2.1 Principle of Pipeline

In the simplest form of pipeline parallelism [4], a sequence of N_s *stages* process a sequence of *inputs* to produce a sequence of *outputs* (Fig. 1). Each input passes through each stage in the same order, and the different inputs are processed one after another (a stage cannot process several inputs at the same time). Note that the internal activity of a stage may be parallel, but this is transparent to our model. In the remainder of the paper we use the term "processor" to denote the hardware responsible for executing such activity, irrespective of its internal design (sequential or parallel).

We consider this application in the context of computational Grids, and so we want to map this application onto our computing resources, which consist of a set of potentially heterogeneous processors interconnected by an heterogeneous network.

Considering the pipeline application in the eSkel library [4], we focus here on the function `Pipeline1for1`, which considers that each stage produces exactly one output for each input.

2.2 Pipeline Model

To model this algorithmic skeleton Grid application, we decompose the problem into the stages, the processors and the network. The model is expressed in Performance Evaluation Process Algebra (PEPA) [9].

The Stages

The first part of the model is the *application model*, which is independent of the resources on which the application will be computed. We define one PEPA component per stage. For $i = 1..N_s$, the component $Stage_i$ works sequentially. At first, it gets data (activity $move_i$), then processes it (activity $process_i$), and finally moves the data to the next stage (activity $move_{i+1}$).

$$Stage_i \stackrel{def}{=} (move_i, \top).(process_i, \top).(move_{i+1}, \top).Stage_i$$

All the rates are unspecified, denoted by the distinguished symbol $\top$, since the processing and move times depend on the resources where the application is running. These rates will be defined later, in another part of the model.

The pipeline application is then defined as a cooperation of the different stages over the $move_i$ activities, for $i = 2..N_s$.

The activities $move_1$ and $move_{N_s+1}$ represent, respectively, the arrival of an input in the application and the transfer of the final output out of the pipeline.

They do not represent any data transfer between stages, so they are not synchronizing the pipeline application. Finally, we have:

$$Pipeline \stackrel{def}{=} Stage_1 \underset{\{move_2\}}{\bowtie} Stage_2 \underset{\{move_3\}}{\bowtie} \cdots \underset{\{move_{N_s}\}}{\bowtie} Stage_{N_s}$$

The Processors

We consider that the application must be mapped on a set of N_p processors. Each stage is processed by a given (unique) processor, but a processor may process several stages (in the case where $N_p < N_s$). In order to keep a simple model, we decide to put information about the processor (such as the load of the processor or the number of stages being processed) directly in the rate μ_i of the activities $process_i$, $i = 1..N_s$ (these activities have been defined for the components $Stage_i$).

Each processor is then represented by a PEPA component which has a cyclic behaviour, consisting of processing sequentially inputs for a stage. Some examples follow.

- In the case when $N_p = N_s$, we map one stage per processor:

$$Processor_i \stackrel{def}{=} (process_i, \mu_i).Processor_i$$

- If several stages are processed by a same processor, we use a choice composition. In the following example ($N_p = 2$ and $N_s = 3$), the first processor processes the two first stages, and the second processor processes the third stage.

$$\begin{aligned}&Processor_1 \stackrel{def}{=} (process_1, \mu_1).Processor_1 + (process_2, \mu_2).Processor_1\\&Processor_2 \stackrel{def}{=} (process_3, \mu_3).Processor_2\end{aligned}$$

Since all processors are independent, the set of processors is defined as a parallel composition of the processor components:

$$Processors \stackrel{def}{=} Processor_1 || Processor_2 || \ldots || Processor_{N_p}$$

The Network

The last part of the model is the network. We do not need to model directly the architecture and the topology of the network for what we aim to do, but we want to get some information about the efficiency of the link connection between pairs of processors. This information is given by affecting the rates λ_i of the $move_i$ activities ($i = 1..N_s + 1$).

- λ_1 represents the connection between the user (providing inputs to the pipeline) and the processor hosting the first stage.

- For $i = 2..N_s$, λ_i represents the connection between the processor hosting stage $i - 1$ and the processor hosting stage i.

- λ_{N_s+1} represents the connection between the processor hosting the last stage and the user (the site where we want the output to be delivered).

When the data is "transferred" on the same computer, the rate is really high, meaning that the connection is fast (compared to a transfer between different sites).

The network is then modelled by the following component:

$$Network \stackrel{def}{=} (move_1, \lambda_1).Network + \ldots + (move_{N_s+1}, \lambda_{N_s+1}).Network$$

The Pipeline Model

Once we have defined the different components of our model, we just have to map the stages onto the processors and the network by using the cooperation combinator. For this, we define the following sets of action types:

- $L_p = \{process_i\}_{i=1..N_s}$ to synchronize the $Pipeline$ and the $Processors$
- $L_m = \{move_i\}_{i=1..N_s+1}$ to synchronize the $Pipeline$ and the $Network$

$$Mapping \stackrel{def}{=} Network \underset{L_m}{\bowtie} Pipeline \underset{L_p}{\bowtie} Processors$$

3 Solving the Models

Numerical results can been computed from such models with the Java Version of the PEPA Workbench [8].

The *performance result* that is pertinent for us is the throughput of the $process_i$ activities ($i = 1..N_s$). Since data passes sequentially through each stage, the throughput is identical for all i, and we need to compute only the throughput of $process_1$ to obtain significant results. This is done by adding the steady-state probabilities of each state in which $process_1$ can happen, and multiplying this by μ_1. This result can be computed by using the command line interface of the PEPA workbench, by invoking the following command:

```
java pepa.workbench.Main -run lr ./pipeline.pepa
```

The `-run lr` (or `-run lnbcg+results`) option means that we use the linear biconjugate gradient method to compute the steady state solution of the model described in the file `./pipeline.pepa`, and then we compute the performance results specified in this file, in our case the throughput of the pipeline.

4 AMoGeT: The Automatic Model Generation Tool

We investigate in this paper how to enhance the performance of Grid applications with the use of algorithmic skeletons and process algebras. To do this, we have created a tool which automatically generates performance models for the pipeline case study, and then solves the models and provides to the application significant results to improve its performance.

We describe the tool succinctly and then provide some numerical results for the pipeline application.

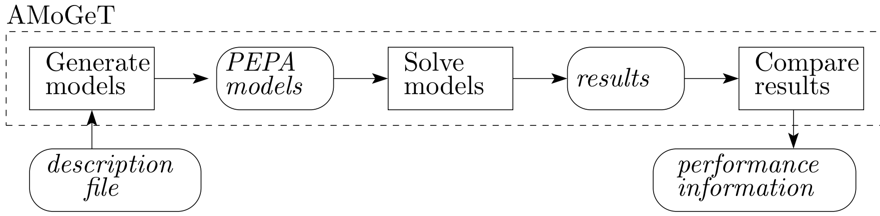

Fig. 2. Principle of AMoGeT

4.1 AMoGeT Description

Fig. 2 illustrates the principle of the tool. In its current form, the tool is a standalone prototype. Its ultimate role will be as an integrated component of a run-time scheduler and re-scheduler, adapting the mapping from application to resources in response to changes in resource availability and performance. The tool allows everything to be done in a single step through a simple Perl script: it generates and solves the models, and then compares the results. This allows us to have feedback on the application when the performance of the available resources is modified.

Information is provided to the tool via a *description file*. This information can be gathered from the Grid resources and from the application definition. In the following experiments, it is provided by the user, but we can also get it automatically from Grid services, for example from the Network Weather Service [12]. We also define a set of candidate mappings of stages to processors. Each mapping specifies where the initial data is located, where the output data must be left and (as a tuple) the processor where each stage is processed. For example, the tuple $(1, 1, 2)$ means that the two first stages are on processor 1, with the third stage on processor 2.

One model is then generated from each mapping, as described in Section 2.2. To compute the rates for a given model, we take into account the number of stages hosted on each processor, and we assume that the work sharing between the stages is equitable. We use also all the other information provided by the description file about the available resources and the characteristics of the application. The models can then be solved with the PEPA workbench, and the throughput of the pipeline is automatically computed (Section 3).

During the resolution, all the results are saved in a single file, and the last step of results comparison finds out which mapping produces the best throughput. This mapping is the one we should use to run the application.

4.2 Numerical Results

We have made some experiments and we give here a few numerical results on an example with 3 pipeline stages (and up to 3 processors). The models that we need to solve are really small (in this case, the model has 27 states and 51 transitions). The time taken to compute these results was very small, being less than one second on this and similar examples involving up to eight pipeline stages. Clearly this is an important property, since we must avoid taking longer to compute a rescheduling than we save in application time by implementing it.

We define `l`i-j to be the latency of the communication between processors i and j, in seconds. We suppose that `l`i-i`=0.0001` for $i = 1..3$, and that there is no need to transfer the input or the output data. We suppose that all stages are equivalent in term of amount of work required, and so we define also the time required to complete a stage on each processor `t`i ($i = 1..3$), if the stage can use all the available processing power (this time is longer when several stages are on the same processor).

We compare the mappings (1,1,1), (1,1,2), (1,2,2), (1,2,1), (1,1,3), (1,3,3), (1,3,1) and (1,2,3) (the first stage is always on processor1), and we only put the optimal mapping in the relevant line of the results table.

Set of results	Parameters						Mapping & Throughput
	l1-2	l2-3	l1-3	t1	t2	t3	
1	0.0001	0.0001	0.0001	0.1	0.1	0.1	(1,2,3): 5.63467
	0.0001	0.0001	0.0001	0.2	0.2	0.2	(1,2,3): 2.81892
2	0.0001	0.0001	0.0001	0.1	0.1	1	(1,2,1): 3.36671
	0.1	0.1	0.1	0.1	0.1	1	(1,2,2): 2.59914
	1	1	1	0.1	0.1	1	(1,1,1): 1.87963
3	0.1	1	1	0.1	0.1	0.1	(1,2,2): 2.59914
	0.1	1	1	1	1	0.01	(1,3,3): 0.49988

In the first set of results, all the processors are identical and the network links are really fast. In these cases, the best mapping always consists of putting one stage on each processor. If we double the time required to complete a stage on each processor (busy processors), the resulting throughput is divided by 2, since only the processing power has an impact on the throughput.

The second set of results illustrates the case when one processor is becoming really busy, in this case processor3. We should not use it any more, but depending on the network links, the optimal mapping may change. If the links are not efficient, we should indeed avoid data transfer and try to put consecutive stages on the same processor.

Finally, the third set of results shows what happens if the network link to processor3 is really slow. In this case again, the use of the processor should be avoided, except if it is a really fast processor compared to the other ones (last line). In this case, we process stage2 and stage3 on the third processor.

5 Conclusions

In the context of Grid applications, the availability and performance of the resources changes dynamically. We have shown through this study that the use of skeletons and performance models of these can produce some relevant information to improve the performance of the application. This has been illustrated on the pipeline skeleton, which is a commonly used algorithmic skeleton. In this case, the models help us to choose the mapping, of the stages onto the processors, which will produce the best throughput. A tool automates all the steps to obtain the result easily.

We are currently working at getting the performance information needed by the tool from the Grid and from the application, to make it more realistic. Moreover, some detailed results on the timing of the tool will be provided. This approach will also be developed on some other skeletons so it may be useful for a larger class of applications. However this first case study has already shown that we can obtain relevant information and that we have the potential to enhance the performance of Grid applications with the use of skeletons and process algebras.

References

1. M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.
2. R. Biswas, M. Frumkin, W. Smith, and R. Van der Wijngaart. Tools and Techniques for Measuring and Improving Grid Performance. In *Proc. of IWDC 2002*, volume 2571 of *LNCS*, pages 45–54, Calcutta, India, 2002. Springer-Verlag.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989. http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz.
4. M. Cole. eSkel: The **e**dinburgh **Ske**leton **l**ibrary. Tutorial Introduction. *Internal Paper, School of Informatics, University of Edinburgh*, 2002. http://homepages.inf.ed.ac.uk/mic/eSkel/.
5. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *To appear in Parallel Computing*, 2004.
6. I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
7. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12):1753–1772, 2002.
8. N.V. Haenel. User Guide for the Java Edition of the PEPA Workbench. *LFCS, University of Edinburgh*, 2003. http://homepages.inf.ed.ac.uk/s9905941/.
9. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
10. N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *JPDC*, 63(5):551–563, May 2003.
11. F.A. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, 2002.
12. R. Wolski, N.T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.