Self-Adaptation of Parallel Applications in Heterogeneous and Dynamic Architectures*

Samir Jafar†, Laurent Pigeon‡, Thierry Gautier and Jean-Louis Roch
Projet MOAIS(CNRS/INPG/INRIA/UJF) – Laboratoire ID-IMAG (UMR 5132)
51 avenue Jean Kuntzmann – 38330 Monbonnot – FRANCE
Email:(Samir.Jafar, Laurent.Pigeon, Thierry.Gautier, Jean-Louis.Roch)@imag.fr

Abstract

In this paper a mechanism for adaptation of parallel computation is defined for data flow computations in dynamic and heterogeneous environments. Our mechanism is especially useful in massively parallel multithreaded computations as found in cluster or grid computing. By basing the state of executions on a data flow graph, this approch shows extreme flexibility with respect to adaptation of parallel computation induced by application. This adaptation reflects needs for changing runtime behavior due to time observable parameters. Specifically, it allows an on-line adaptation of parallel execution in dynamic heterogeneous systems. We have implemented this mechnism in KAAPI *(Kernel for Adaptative and Asynchronous Parallel Interface) and experimental results show the overhead induced is small.*

1 Introduction

A current trend in parallel computing is to be able to execute parallel applications with communication between tasks on a large scale distributed and heteogeneous platform including SMP (Symmetric Multi Processor) nodes. Grid, cluster and peer to peer architectures are gaining in popularity for scientific computing applications. Such architectures are well known to be heterogeneous and highly dynamic in content and load, *i.e.* nodes are continuously joining and leaving the system.

Regarding parallel applications, this feature is a challenging problem to fully exploit the whole parallelism induced by the application. This paper deals with adapting the execution of parallel applications to the executing environment in order to reach high performances.

In this paper we present our framework for writing grid applications in order to adapt them on large heterogeneous and dynamic grid. Our work extends previous approaches [1, 11] by making transparent to the user the mechanism of adaptation. The idea is based on computational reflection [1, 8] of the system: At runtime the casually connected representation of the system is a data flow graph between the tasks of the future of the execution [4]. It allows to take into account the structure of the computation, including communications between tasks (*i.e.* jobs) of the application to makes self-adaptation of parallel execution.

Within the context of this research this graph is dynamic, *i.e.* it changes during runtime as the result of task creations or terminations. This representation is maintained by our framework at low cost and it defines the global state of the application's execution. Based on this execution model, we define the online adaptation of a parallel execution as the solution of the two following problems: leaving of computational resources is solved by using fault-resilience mechanisms [6, 7]; and adapting the performance to a variation (joining or leaving) of resources is equivalent to solve an online scheduling problem.

This paper presents our technical choices to implement self-adaptation of parallel applications in heterogeneous and dynamic architecture. Thanks to KAAPI runtime and to its abstract representation of the execution, a fully useful middleware can manage and run large applications on a grid oriented architecture under dynamic constraints. This paper also proposes to evaluate the overhead of a such approach in the context of an online scheduling of recursive application by workstealing with respect to variation of the number of resources.

The second section of the paper presents related works on adaptability for parallel computing. Section 3 presents our execution model for parallel and distributed applications. In section 4, we present our approach for self-adaptation of parallel applications which is based on the graph of dependency associated to a parallel program. In section 5, we present an implementation of this mechanism on KAAPI and experimental results will be presented.

*This work is supported by a AHA project.
†This author is supported by a grant of the Syrian Government
‡This author is supported by the IFP (French Institute of Petroleum)

2 Related Work

In recent years, self-adaptation of parallel and distributed applications have attracted more and more attention for high performance of execution in a dynamic and heterogeneous platform. The study of self-adaptation mechanisms suited to such platforms is now an active research area [2, 13, 3, 12, 1, 11]. These studies differ in the kind of adaptation with respect to computing environments, software or performance.

Authors of [1, 11] have studied self adaptively based on either fine (message passing) or coarse (component) grain respectively. Both projects study frameworks to adapt the workload of applications to the computational environment with respect to a given estimation of the performances of the grid. The adaptation mechanism relies on an operation to migrate a component or a process. The points where the application might migrate are explicitly provided by the programmer of the application. A controller takes decision to adapt the execution of the application regarding the predicted times with respect to the gathered execution times using monitoring and analysis tools provided by others parts of their framework.

Our work extends previous approaches by making transparent to the user:

1. The definition of migration points in an application.

2. The re-scheduling of the application to adapt the performance on variation of resources.

The idea is based on computational reflection [1, 8] of the system: At runtime the casually connected representation of the system is a data flow graph between the tasks of the future of the execution [4]. It allows to take into account the structure of the computation, including communications between tasks (*i.e.* jobs) of the application to solves (1) and (2).

3 Execution Model

KAAPI is a middleware that allows to develop an application in such manner that the scheduling is a plugin: changing the scheduling algorithm does not require any modification of the application. The architecture of the middleware is sketched in figure 1. The key point in the design of KAAPI is its abstract representation of application as a data flow graph. This representation was firstly design because it is well suited for scheduling algorithm. In [6, 7] we extend the use of this representation to add fault-tolerance support in KAAPI.

The view of the application through the middleware is a data flow graph that describes the computation. This representation is the input of high level algorithm such that scheduling algorithms or fault-tolerence protocols. The runtime environment is view by KAAPI through sensors that monitor the resources and by decision such that instantiation of new process.

3.1 Data flow graph representation

At the base of the execution model is the macro data flow model. A data flow graph [10] allows for a natural representation of a parallel execution, and it can be exploited to achieve fault-tolerance [9]. By the principle of data flow, tasks become ready for execution upon availability of their input data.

Definition 1 *A data flow graph is defined as a directed graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of vertices and $\mathcal{E}$ is a set of edges representing precedence relations between vertices. The vertex set consists of computational tasks, as seen in the traditional context of task scheduling, and the edge set represents the data dependencies between the tasks.*

Within the context of this research G is a dynamic graph, *i.e.* it changes during runtime as the result of task creations or terminations. The target environment for multithreaded computations with data flow synchronization between threads is the Kernel for Adaptive, Asynchronous Parallel Interface (KAAPI) [6], implemented as a C++ library. The library also contains an high level API called Athapascan [5] and it is able to schedule programs at fine or medium granularity in a distributed environment [4].

3.2 Definition of execution state

The runtime support in KAAPI defines several objects. The data flow graph as presented above may be partitioned due in order to distributed the work-load among the processors. A thread of control is in charge of executing tasks in a partition of the data flow graph. A process may handles several threads and a set of communicating processes represents the distributed execution of application.

The state of the application's execution is the states of all processes. In [6, 7] we have presented how to define the global state of the application's execution by reducing it to the local states of the data flow graph handle by each process. Thus, the macro data flow graph to define the state of the application's execution. The graph is a representation of the computational tasks to be carried out along with the associated data, which constitute the inputs and outputs. The data flow is dynamic

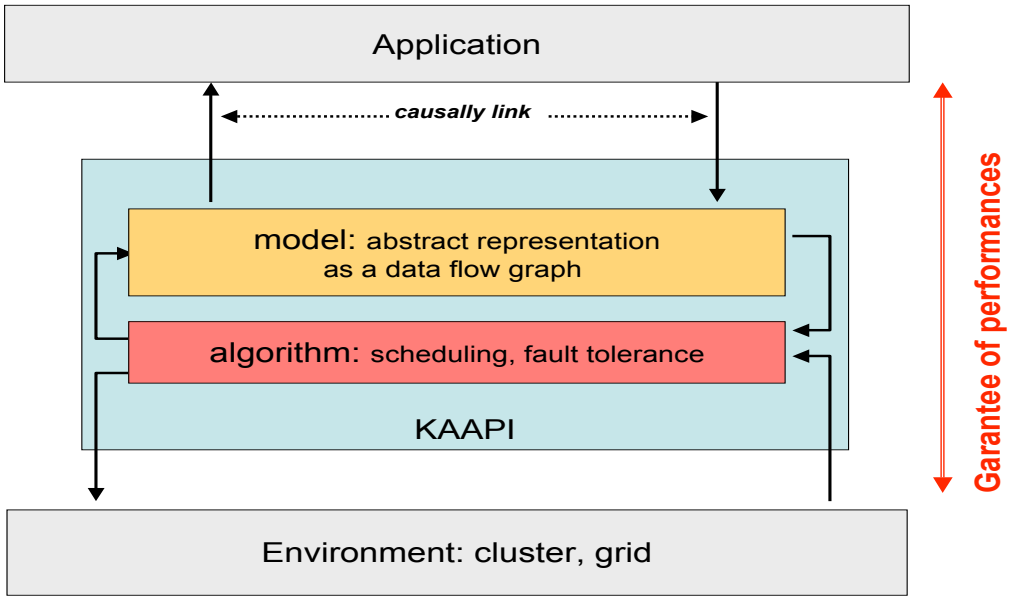

Figure 1: Architecture of KAAPI middleware.

is platform-independent. As a result the graph or portions of it can be moved across heterogeneous platforms during execution. Formally, at any instance of time t, the macro data flow graph G describes a platform-independent, and thus portable, consistent global state of the execution of an application. Whereas graph G is viewed as a single virtual data flow graph, its implementation is in fact distributed. Specifically, each process i contains and executes a subgraph G_i of G.

4 Self-Adaptation of parallel applications

In a dynamic platform two important changes can occur during the execution of application: the departure and the arrival of resources with respect to the application. Next sections describe the adaptation mechanism for this both cases. The implementation of these adaptation mechanism is done in the "algorithm" box described in the architecture of KAAPI in figure 1.

4.1 Adaptation to the departure of resources

One can classify the departure of resources in two classes: involuntary or voluntary departures. The involuntary departure happens in case of failure of resource. The application does not have any information about the date of the failure and should anticipate it. The key idea to process the two previous cases is to define a portable checkpoint of process. This checkpoint must be platform-independent and one does not need to allocate a new resource to restart the leaving resource from its last checkpoint. For instance, process on alive resource may integrate the computation stored in the checkpoint to its own running computation. Fault-tolerance protocol is applied in this case such as those describe in [6, 7]. The mechanisms is based on portable and distributed checkpoint of the computation state of parallel application and allows local recovery of the only crashed resources. The departure of a resource is called *voluntary* when the system knows the date when this resource will leave the computation. This is the case when using a batch system to reserve machines with bounded time. In case of voluntary departure of resource, the idea of "migration of computation" is applied: just before the date of the departure, a checkpoint is produced, and the workload and data is moved to another resource.

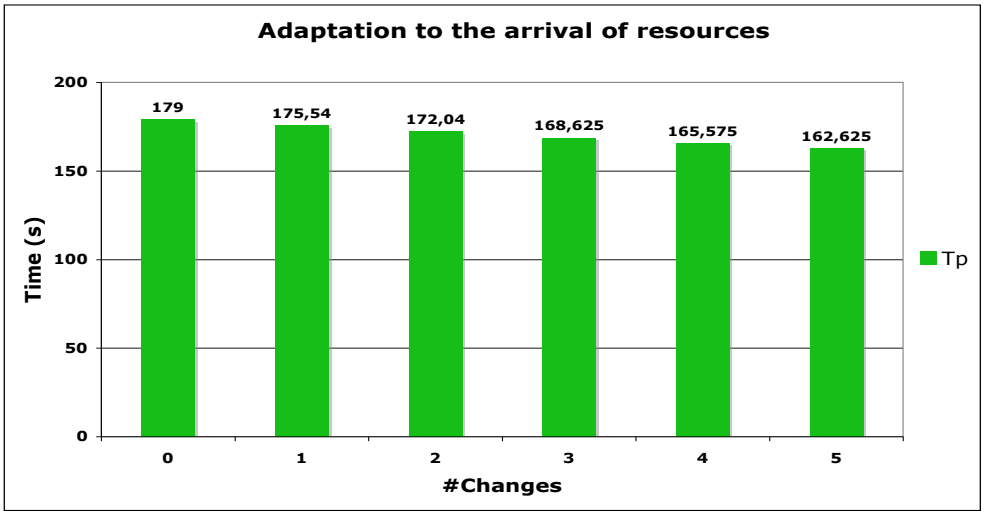

Figure 2: Efficiency of the adaptation mechanism to the arrival of new resources.

4.2 Adapting to the arrival of resources

In order to reach some high performance computations, the work has to be load balanced using a scheduling algorithm that computes the distribution of the work. This is important if several new resources may be used to gain in performance. In this paper we study two main scheduling algorithms classes. The former is based on a dynamic scheduling by work-stealing. The later uses static graph scheduling algorithms.

The main concept behind work-stealing algorithm is that an idle resource steals some work from an overloaded node. Based on this remark, the addition of computing resources is trivial: the idle resource begins by stealing work. The deletion of resources involves the redistribution of checkpoints to others alive resources using our fault-resilience mechanism.

Our second scheduling algorithms class is best suited in case of numerical simulation applications where the whole data flow graph of the application is known before the execution of the tasks (note that with KAAPI this is done at runtime). Using static graph scheduling algorithms can produce a quasi-optimal scheduling of the data flow graph, taking into account communication overheads. Such a method usually allows to reuse the same schedule for several steps until the workload becomes unbalanced. In case of detection or deletion of resources, our approach is to recompute a schedule of the tasks with the new number of resources. This possible because KAAPI maintains a causally connected representation of the application's execution. The adaptation mechanism load checkpoint to reconstruct the data flow graph of the execution to be re-scheduled. Thus, at any time, it is possible to reconfigure a running computation and to optimise the performances of execution by using the results of a static scheduling algorithm.

5 Experimental results

We have implemented the self-adaptation mechanism on KAAPI [7, 6]. The library is able to schedule fine/medium size grain program on distributed machine. The performance and overhead of the self-adaptation mechanism were experimentally determined for the Fibonacci Banchmark, *i.e.* Fibonacci number computation, which was parallelized in KAAPI.

The experiments were conducted on a cluster of Grid5000 architecture[1]. The cluster consists of 32 nodes interconnected by a 100Mbps Ethernet network. Each node features two processors (2000 MHz) and 2 GB of main memory.

The execution on a single processor, called T_s, ends

[1] http://www.grid5000.fr/

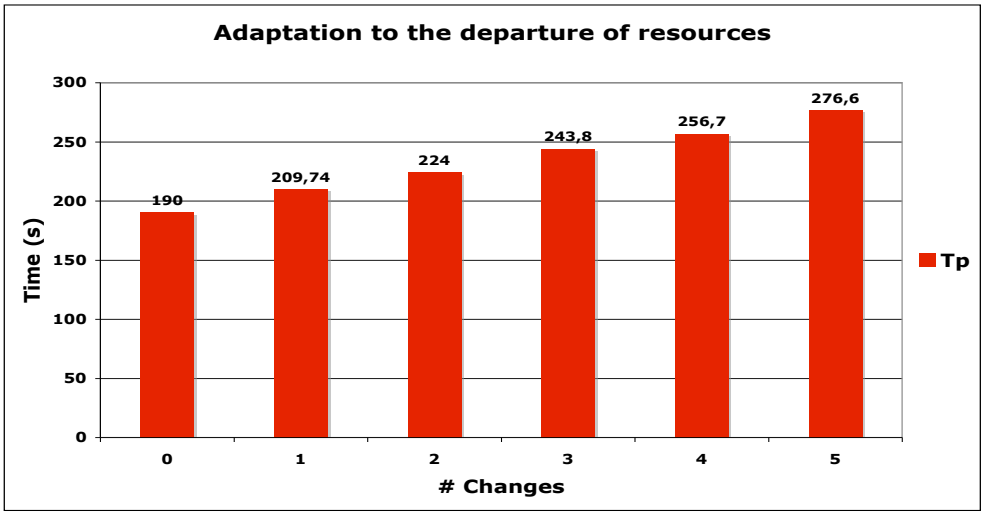

Figure 3: Efficiency of the adaptation mechanism to the departure of resources.

in 6605 seconds. Using $p = 40$ processors the completion time, called T_p, is 179 seconds Figure 2 shows the times of the execution when new resources are joining the computation. The experiment is the following. For the i-th bar in the figure, the execution starts on the cluster with 40 processors, and each $20 \times i,\ i = 0, ..., 5$ seconds from the start we add a new processor. The time is decreasing when new processor are joining the computation. The observed efficiency in resource utilization is good ($> 90\%$).

Next experiment presents the impact of our adaptation mechanism to the departure of resources. Figure 3 reports times of the following experience. The execution starts on the cluster with 40 processors, after 20 seconds from the start we kill one processor and the execution continues on 39 processors, after 40 seconds we kill a second processor and the execution continues on 38 processors, etc. To drive this experiment fault-tolerance protocol described in [7] was used.

The execution on a single processor with checkpointing, using one checkpoint server, every 10 seconds is 6905 seconds. On $p = 40$ processors the completion time is 190 seconds. There is a small overhead due to the checkpointing which respect to the times given above. A depicted in the figure, the runtime is increasing with respect of the number of processors that have leaving the computation. But it noticeable better in comparison in the loss of full work if the computation of the leaving processor was not checkpointed.

6 Conclusion

In order to address performance of large parallel applications we have introduced self-adaptation mechanism for data flow applications in heterogeneous and dynamic platform. The state of the parallel application is represented in a portable fashion utilizing data flow graph and is a first class object on which the adaptation algorithm acts. Due to the casually connected representation of the data flow graph with the application, any modification of the representation is traduced on modification on the behavior of the execution. Moreover, the abstract representation is a platform-independent description of the application state, which makes it possible to adapt in a dynamic and heterogeneous cluster or grid. The overhead associated with this mechanism was shown to be small, to the point of being negligible in case of resources addition.

References

[1] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for grid computing. In

EGC, pages 538–547, 2005.

[2] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and lapack for clusters, 2003.

[3] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications, 2002.

[4] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In IEEE, editor, *PACT'98*, pages 88–95, Paris, France, October 1998.

[5] R. Revire J. L. Roch, T. Gautier. Athapascan: Api for asynchronous parallel programming. Technical Report RT-0276, `www-id.imag.fr/software/ath1`, Projet APACHE, INRIA, February 2003.

[6] S. Jafar, T. Gautier, A. Krings, and J-L. Roch. A checkpoint/recovery model for heterogeneous dataflow computations using work-stealing. In *Proceedings of EuroPar'05*, Portugal, August 2005.

[7] S. Jafar, A. Krings, T. Gautier, and J-L. Roch. Theft-induced checkpointing for reconfigurable dataflow applications. In *Proceedings of the IEEE Electro/Information Technology Conference EIT*, U.S.A., May 2005.

[8] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[9] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Proceedings 15 th Symposium on Reliable Distributed Systesm*, pages 2–11, 1996.

[10] J. Silc, B. Robic, and T. Ungerer. *Asynchrony in parallel computing: from dataflow to multithreading*, pages 1–33. Nova Science Publishers, Inc., 2001.

[11] Sathish S Vadhiyar and Jack J Dongarra. Self adaptivity in grid computing. *Concurrency and Computation*, 17(24):235–257, 2005.

[12] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. pages 46–46, 2000.

[13] R. Clint Whaley and Jack Dongarra. Self adapting linear algebra algorithms and software. In *SC'98: High Performance Networking and Computing*, 1998.