AN OBJECT-ORIENTED MODEL FOR ADAPTIVE HIGH-PERFORMANCE COMPUTING ON THE COMPUTATIONAL GRID

THÈSE N^{O} 3079 (2004)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut des systèmes informatiques et multimédias

SECTION D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Tuan Anh NGUYEN

Bachelor of Computer Sciences, University of Technology, Ho-Chi-Minh city, Viêt-Nam
et de nationalité vietnamienne

acceptée sur proposition du jury:

Prof. G. Coray, directeur de thèse
Prof. B. Chopard, rapporteur
Dr P. Kuonen, rapporteur
Prof. R. Perrot, rapporteur
Prof. J.-P. Thiran, rapporteur

Lausanne, EPFL
2005

Abstract

The dissertation presents a new parallel programming paradigm for developing high performance (HPC) applications on the Grid. We address the question "How to tailor HPC applications to the Grid?" where the heterogeneity and the large scale of resources are the two main issues. We respond to the question at two different levels: the programming tool level and the parallelization concept level.

At the programming tool level, the adaptation of applications to the Grid environment consists of two forms: either the application components should somehow decompose dynamically based on the available resources; or the components should be able to ask the infrastructure to select automatically the suitable resources by providing descriptive information about the resource requirements. These two forms of adaptation lead to the *parallel object model* on which resource requirements are integrated into *shareable distributed objects* under the form of *object descriptions*. We develop a tool called ParoC++ that implements the parallel object model. ParoC++ provides a comprehensive object-oriented infrastructure for developing and integrating HPC applications, for managing the Grid environment and for executing applications on the Grid.

At the parallelization concept level, we investigate the *parallelization scheme* which provides the user a method to express the parallelism to satisfy the user specified time constraints for a class of problems with known (or well-estimated) complexities on the Grid. The parallelization scheme is constructed on the following two principal elements: the *decomposition tree* which represents the multi-level decomposition and the *decomposition dependency graph* which defines the partial order of execution within each decomposition. Through the scheme, the parallelism grain will be automatically chosen based on the available resources at run-time. The parallelization scheme framework has been implemented using the ParoC++. This framework provides a high level abstraction which hides all of the complexities of the Grid environment so that users can focus on the "logic" of their problems.

The dissertation has been accompanied with a series of benchmarks and two real life applications from image analysis for real-time textile manufacturing and from snow simulation and avalanche warning. The results show the effectiveness of ParoC++ on developing high performance computing applications and in particular for solving the time constraint problems on the Grid.

Résumé

Cette thèse présente un nouveau paradigme pour le développement d'applications de calcul de haute performance (HPC : High Performance Computing) dans des environnements de type GRILLE (GRID). Nous nous intéressons plus particulièrement à adapter les applications HPC à des environnements où le nombre et l'hétérogénéités des ressources est importantes comme c'est le cas pour la GRILLE. Nous attaquons ce problème sur deux niveaux : au niveau des outils de programmation et au niveau du concept de parallélisme.

En ce qui concerne les outils de programmation, l'adaptation à des environnements de type GRILLE est de deux formes : les composants de l'applications doivent, d'une manière ou d'une autre, se décomposer dynamiquement en fonction des ressources disponibles et les composants doivent être capables de demander à l'infrastructure disponible de choisir automatiquement des ressources adaptées à leur besoin; pour cela elles doivent être capables de décrire leur besoin en terme de ressources nécessaires. Ces deux formes d'adaptation nous ont conduit à un modèle d'objets parallèles. Grâce à ce modèle nous pouvons exprimer les exigences en terme de ressources sous la forme de descriptions d'objets intégrées dans un modèle d'objets distribués partageables. Nous avons développé un outil appelé ParcoC++ qui implémente le modèle des objets parallèles. ParoC++ fourni l'infrastructure nécessaire pour développer et intégrer des applications HPC, pour gérer un environnement GRID afin d'exécuter une telle application.

Au niveau du concept de parallélisme, nous avons introduit la notion de schéma de parallélisation (parallelization scheme) qui fourni à l'utilisateur un moyen d'exprimer le parallélisme afin de satisfaire à des contraintes de temps d'exécution pour des problèmes dont la complexité est connue ou peut être estimée. La notion de schéma de parallélisation est construite sur les principes suivants : l'arbre de décomposition qui représente les différents niveaux de décomposition du problème et le graphe de dépendance de la décomposition qui défini un ordre partiel d'exécution pour une décomposition donnée. Grâce à ces notions nous pouvons automatiquement adapter le grain du parallélisme aux ressources choisies au moment de l'exécution. A l'aide de ParoC++ nous avons réalisé un environnement intégrant la notion de schéma de parallélisation. Cet environnement fourni un haut niveau d'abstraction qui cache à l'utilisateur la complexité de la GRILLE de manière à ce qu'il puisse se concentrer sur la " logique " de son problème.

Pour valider notre environnement, nous avons effectué une série de tests de performance et nous l'avons utilisé pour réaliser deux grosses applications : une application industrielle dans le domaine du traitement d'image et une application pour la recherche dans le domaine de la prédiction des avalanches. Les résultats montrent que ParoC++ est un outil adéquat pour le développement d'applications HPC ayant des contraintes de temps d'exécution et s'exécutant sur une GRILLE.

Acknowledgements

Five-year studying and working in Switzerland has been the source of great pleasure for me and I would like to acknowledge the people who helped and supported me during this period.

I am most indebted to Professor Giovanni Coray and Professor Pierre Kuonen for their valuable guidance and encouragement. Their vision, their creativeness, their enthusiasm and their personalities have inspired my life and my research. I am also grateful to them for giving me complete freedom in my research work although they have always been there to help me when necessary; giving me a huge support since the first day I was in Switzerland. Working with them is extremely enjoyable and rewarding experience.

One of the most beautiful experiences of my research in Switzerland is traveling and working on different projects where I met great people from different fields of science. I express my gratitude to Professor Jean-Philippe Thiran for his help and his guidance in the field of image processing. I am thankful Prof. Bastien Chopard for his precious comments to improve the quality of text of the thesis. I am also thankful to Dr. Michael Lehning for his comments and his help in my work. I learn from him about the snow process and the snow research which I would never experience in Vietnam.

The Department of Information Technology at the Ho Chi Minh city University of Technology is the place where I have had long time studying and working. I express my gratitude to professors and colleagues of the department for their help and their collaboration. In particular, I am greatly thankful to Professor Nguyen Thanh Son and Professor Phan Thi Tuoi who have encouraged and guided me on my research career.

My first two years in Switzerland was supported by a scholarship from the Swiss Federal Commission for Scholarships. I gratefully acknowledge them for giving me an opportunity to study and to know about the people and the country of Switzerland.

I appreciate my friends and colleagues at EIA-FR for their generous support, especially Jean-François Roche and Dominik Stankowski. I have the company of many people during this period. I also take this opportunity to thank them for their fruitful friendship and their help. In particular, I am thankful to Nguyen Ngoc Anh Vu, Cao Thanh Thuy, Nguyen Ngoc Tuan, Vo Duc Duy, Vu Xuan Ha, Le Lam Son, Le Quan, Vu Minh Tuan and Do Tra My for their great encouragement and support.

I owe deeply my parents, my grand father and my sister. They are always a bright light of my life and I would like to dedicate this dissertation to them as a gift for their constant support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Motivation

Parallel high performance computing has been an active subject of research during the last decades. With the development of microprocessor techniques and later the rapid growth of the Internet, the purpose and the methodology of high performance computing (HPC) have been changed. Old fashion HPC applications built on high-cost, high-power consumption and special purpose systems start to be replaced by applications running on low-cost, highly integrated, high-speed processors and fast Ethernet and/or Internet communications. Computing power is not any more centralized but it is rather geographically distributed on the Internet. Grid computing, a new concept, is emerged by coordinating HPC computing and data resources (computers, supercomputers, workstations, storage,...) over the world to form a world-scale virtual supercomputer. This will lead to the need to build new system software, tools to support: multi-level parallelism, large scale HPC applications with complex data structures, complex, dynamic, volatile and unpredictable environments with high heterogeneity.

The emerging of computational grid [29, 31] and the rapid growth of the Internet technology have created new challenges for application programmers and system developers. Special purpose massively parallel systems are being replaced by loosely coupled or distributed general-purpose multiprocessor systems with high-speed network connections. Due to the natural difficulty of the new distributed environment, the programming methodologies that have been used before need to be rethought.

Many system-level toolkits such as Globus [28], Legion [38] have been developed to manage the complexity of the distributed computational environment. They provide services such as resource allocation, information discovery, user authentication, etc. However, since the user must deal directly with the computational environment, developing applications using such tools still remains tricky and time consuming.

At the programming level, there still exists the question of achieving high performance computing (HPC) in a widely distributed heterogeneous computational environment. Some efforts have been spent for porting existing tools such as Mentat Programming Language (MPL) [41], MPI [27] to the computational grid environment. Nevertheless, the support for adaptive usage of resources is still limited in some specific services such as network bandwidth and real-time scheduling. MPICH-GQ [69], for example, uses quality of service (QoS) mechanisms to improve performance of message passing. However, message passing is a quite low-level library that the user has to explicitly specify the send, receive and synchronization between processes and most of parallelization tasks are left to the programmer.

The above difficulties lead to a quest for a new programming paradigm and a new programming model for developing HPC applications on the Grid. We will go a step further to develop a parallelization model that allows the user to tackle time constrained problems-problems that require the solution be obtained within a user specified time interval.

1.2. Contributions of the dissertation

This dissertation addresses the question: *"How to tailor applications with a desired performance to the Grid?"*. The answer is obtained at two different levels, following the meaning of "desired performance": the low-level performance in which the desired overall performance is constituted by the desired performance of different application components; and the high-level performance in which the user requests explicitly the overall application performance in terms of the required computation time.

The main contributions of this dissertation are: ***a requirement-driven object-oriented model*** to address the low-level performance of application components for the Grid; ***the parallelization scheme*** to solve time constrained problems on the Grid; and the ***ParoC++ tool*** which provides a new programming paradigm based on the object-oriented model for the Grid.

1.2.1. The parallel object model and the ParoC++ system

The contributions in this part include:

- The parallel object model that generalizes the traditional sequential object model by adding the resource requirements, different method invocation semantics, remote distribution and transparent resource allocation to each parallel object. Parallel object provides a new programming paradigm for high performance computing applications. According to the model, parallel objects are the elemental processing units of the application.

- ParoC++ programming language that extends C++ to support the parallel object model. ParoC++ adds some extra keywords to C++ allowing the programmer to implement:

  - Parallel object classes.
  - Object descriptions (ODs) that describe the resource requirements for each parallel object. OD is used to address the application adaptation to the heterogeneous environment.
  - The inter-object and intra-object communication.
  - The concurrency control mechanism inside each parallel object.
  - Exception mechanism for distributed parallel objects.

- ParoC++ compiler to compile the ParoC++ source codes.

- ParoC++ runtime system to execute ParoC++ applications. The ParoC++ design principle is to glue other low-level distributed toolkits for executing HPC applications. The ParoC++ run-time architecture is an abstract architecture that allows the integration of new system into the existing one in the plug-and-play flavor.

  - ParoC++ execution model that describes the binary organization structures of a ParoC++ application and how a typical application operates.
  - ParoC++ service model that introduces the application scope service type.
  - ParoC++ resource discovery model- a fully distributed resource discovery for parallel object allocation. This model takes into account issues of fault-tolerance and dynamic information states of the Grid.
  - ParoC++ object manager service to allow dynamic parallel object allocation.
  - A guideline for the integration of other low-level toolkits into the ParoC++ system with an example of Globus integration.

- Passive data access method using ParoC++. The method provides an efficient way to access data with the ability to predict, to partially process and to synthesize data from multiple data sources.

- Set of experiments and test cases to demonstrate different aspects of the ParoC++ system.

1.2.2. Parallelization scheme for problems with time constraints

In this part, we will address the time constraint issues for a class of problems with known complexities on the Grid. First, we provide the programmer a *parallelization scheme* to describe the time constrained problems:

- A way the user decomposes his time constrained problem and the relationship between each decomposition.

- Algorithms to find a suitable solution (solution whose computation time satisfies the time constraint) on the Grid.

Then, we develop an object-oriented framework that uses ParoC++ to implement the parallelization scheme. The user can concentrate on decomposing the problem and defining the relationship of sub-problems in each decomposition. The framework will dynamically solve the problem with a suitable grain of parallelism in order to satisfy the required time constraint based on the currently available resources inside the environment.

Finally, we discuss some experiments and a test case using the framework.

1.3. Dissertation outline

The rest of the dissertation is divided into three parts: the first part from chapter 2 to chapter 4 is the theory part of the dissertation. We first present the state-of-the-art of the Grid computing and its challenges in chapter 2. Then we will move on to chapter 3 to present our parallel object model which provides programmers an object-oriented programming paradigm based on requirement-driven objects for high performance computing. Expressing the parallelism in time constrained applications is addressed through the parallelization scheme that we will present in chapter 4.

Part 2, from chapter 5 to chapter 8 discusses the ParoC++ programming system which implements the parallel object model and a framework for developing time constrained applications. We discuss different features of the ParoC++ system from programming language aspects (chapter 5), programming methods using ParoC++ to improve data movement in HPC (chapter 6), to the ParoC++ infrastructure and the integration with other environments with Globus toolkit as an integration example (chapter 7). Chapter 8 deals with developing time constrained applications, and real-time applications in particular. Based on the parallelization scheme in chapter 4 and the ParoC++ system in chapter 5, we develop a ParoC++ framework for solving problems with time constraints and illustrate how to use this framework for solving such problems on the Grid.

Part 3 presents the experiment results of the ParoC++ system and the parallelization scheme that we described in part 2. Chapter 9 describes the benchmarks of the ParoC++ system and some small experiments on ParoC++ as well as on an emulated-time constrained application with the framework. Chapter 10 starts the first test case of ParoC++ on the pattern and defect detection system for textile manufacturing. Chapter 11 gives a demonstration of how to use ParoC++ not only as a tool to parallelize but also the tool to integrate and to manage a complex system of snow modeling, run off and avalanche warning system. The experiment part ends with chapter 12 as the last test case on how to use the parallelization scheme for a real-time image analysis application.

Chapter 13 is the conclusion of the dissertation.

Part I

State-of-the-art and the parallel object model

Chapter 2

Background and related work

In this chapter, we will review the state-of-the-art of Grid computing. We focus on two subjects: the supporting infrastructures and the programming models. From the infrastructure aspects, after introducing the Grid concepts, we will examine the evolution of the Grid and some well-known Grid supporting toolkits. Currently, there is no programming model particularly designed for the Grid. Most of programming models used on the Grid are extended from traditional programming models. Therefore, for programming models, we will present some practical programming models for distributed environments and their use on the Grid.

2.1. The computational Grid

2.1.1. Grid definition

The term "computational Grid" (or the Grid for short) emerged in the mid of 1990s has been used to refer to the infrastructure for advanced science and engineering. By borrowing the idea of the electric power grid, Ian Foster and Carl Kesselman, the two pioneers in Grid computing, give the definition of computational Grid in [29]: "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities". The definition mentions different characteristics of the Grid. The infrastructure of the Grid means we need to deal with a large confederation of resources which can be the computing capabilities such as computers, supercomputers, clusters, etc.; data storages, sensors or even human knowledge involving in the computational environment to provide services. Dependable service means the user who uses the Grid should be guaranteed on the quality, the reliability and the stability of the services that constitute the Grid. The resources in the Grid are heterogeneous that can be differed on hardware architectures, hardware capacities, operating systems, software environments, security policies, etc. The Grid user should be able to gain a consistent access via some standard interfaces to the Grid service regardless of such differences. The resources tend to be distributed over the Internet and are connected with high-speed connections, so pervasive access enables users to access to the service no matter where they are located or what environments they are working on. Finally, inexpensive access, despite not a fundamental characteristic, is also an important factor in wide spreading the use of the Grid like that of the electric power Grid today.

2.1.2. Domains of Grid computing

One question we need to answer in order to understand the Grid is "what is it used for?". The application field of the Grid is variety in science and engineering. The Grid covers 4 categories of applications: collaborative engineering, data exploitation, high-throughput computing and distributed supercomputing [29].

In *collaborative engineering*, scientists at different sites work together interactively through the Grid, doing some experiments or discussing the results in a "virtual laboratory" located somewhere else. They can manipulate the virtual device as if the device were located locally at their site. Applications in this category can be virtual reality systems, simulations, visualizations, astronomic observations, etc.

Data exploitation allows scientists to explore and to access a huge volume of data produced by some sources remotely. For instance, experiments in the field of high energy physics at Large Hadron Collider (LHC) [17], the most powerful particle physics accelerator ever constructed at CERN which will be finished in 2007, will produce petabytes of data annually. Nevertheless, for a specific group of scientists, only part of this data really needs to be efficiently accessed and modified while the rest are kept untouched. The amount of data is usually too big to fit into a single storage device. Instead it is likely distributed over several places. Therefore, the Grid can help to manage, to move, to aggregate and to access the data remotely in a secure manner.

High-throughput computing uses the Grid to schedule large numbers of relatively independent tasks on idle resources for solving problems. Making use of free processor cycles over the Internet can lead to a large amount of computations to be performed in order to tackle computational hard problems. However, only problems that can be decomposed into loosely coupled sub-problems with little data exchange between components can benefit from high-throughput computing. The probably most typical example is the use of SETI@Home (Search for Extraterrestrial Intelligence) network [81] to analyze data from space. The user contributes the idle cycles under a screen saver program. In October, 2003, more than 4.7 million users have contributed their cycles and the aggregate performance is more than 60 Teraflops/sec, faster than the most powerful computer ever constructed to date. Folding@home [71, 87, 83] is another example of large-scale high throughput computing to study protein folding process in biology where users donate their CPU time under a screen saver. Since 2000 when the project was started, almost 1 million CPU throughout the world have been used with the accumulated computing power of more than 10000 CPU-year work.

Distributed high performance computing (DHPC) is used to combine the computing power of computers, clusters and supercomputers that are geographically distributed to tackle big problems that can not be solved in a single system. Differ from high-throughput computing, DHPC applications place high requirements on distributed resources such as the peak computing power, the memory size or the external storage. In addition, different computational modules can be tightly coupled that require high speed communication among distributed resources. The Grid services coordinate these distributed resources and may be used as a portal to locate, to reserve and to access remote resources.

2.1.3. Challenges

The Grid is an emerging technology. It has been growing very rapidly during the past few years but it is not mature yet. The Grid computing infrastructure is still in the research phase. At the moment, it is too early to define a standard for the Grid. In order to become a standard, many challenges need to be overcome.

The first challenge is on how to exploit the power of the Grid. Because Grid computing differs from conventional parallel distributed computing in a number of fundamental ways, the programming model and programming methodology should be rethought. Conventional applications based on a resource-centric approach should be changed to the service-centric approach as did the Grid services. Grid applications should adapt to the heterogeneity of the environment. Fault tolerance which is not the major problem in the conventional environment should be carefully taken into account. The success of the Grid also depends on how easily the user can develop and deploy his Grid applications. High level programming tools specially designed for developing Grid applications are not available yet.

Secondly, the connectivity of resources and of application components is also a major concern. We know that Internet is an unreliable and untruthful environment where resources can be attacked by hackers all the time. Firewalls have been established to prevent such attacks. However, these firewalls also prevent the ability to establish direct connections between components. How to enable full scale resource sharing as well as to guarantee the privacy and the security is a technology challenge.

The third challenge is on the scalability of the Grid. Managing resources within a single organization does not usually face with the scalability issue. However, when the geographically distributed resources reach millions and belong to different organizations, an efficient management mechanism becomes a main issue. Current toolkits such as Globus [28] or Legion [38] only address some issues such as security issues and distributed information management. Issues such as resource discovery, resource reservation, self management, fault tolerance still need to be further investigated

Next, we have to deal with how to evaluate the Grid and its applications. At the time being, no suitable method for measuring the efficiency of the Grid and its applications is available. The traditional measurement of system efficiency as the effective performance (e.g. the number of floating point operations per second) over the peak performance of the system is not correct in the Grid. The parallel efficiency measurement of the application as the ratio between the speedup and the number of processors fails to work on the Grid due to the heterogeneous nature of the environment.

Finally, accounting is also an important issue of the Grid system. The wide usage of the Grid will not be able to depend only on the free donation of resources. To guarantee the success of the Grid, it is necessary to have "Grid companies" that can sell their resources. "What is the price policy?" and "how to charge the Grid user for using the resources?" are among the questions needed to be investigated. The answers should be in consensus between the provider and the user.

In this dissertation, we will focus on the challenge of *how to efficiently exploit the power of the Grid for high performance applications* and particularly applications with time constraints through the application adaptation. We will not develop a new metric to measure the parallel efficiency of applications on the Grid but we will consider the *efficiency in our sense as the maximum amount of speedup that an application can gain from the Grid environment and the ability of an application to satisfy the user time requirements.*

2.1.4. Grid evolution

Up to now, the evolution of the Grid goes through the two major phases. The first phase focused on finding the answers for: "Is that feasible to build a Grid infrastructure?" and "Which Grid services are needed inside this infrastructure?". In this phase, major Grid services have been identified and tested: the resource management service, the information service, the security service, etc.. A number of middleware have been built up. Among them, two of the well-known ones are Globus, Legion. GUSTO-a Globus testbed, has been constructed to test the feasibility of the Grid concept. In the year 2000, more than 125 universities and institutions over the world joined the GUSTO testbed with the aggregate computing power of over 5 Teraflops/sec.

The second phase of the Grid evolution is on-going, focusing on the technology challenges such as the portability and the inter-operability of Grid components. The new web technologies such as Web services [16], Java and SOAP [84] have been used in Grid components that improve considerably the operability of the Grid. The emerging of the Open Grid Service Architecture (OGSA) [30] from the Global Grid Forum is an important step toward the standardization of Grid components and services. OGSA is based on Web service technologies for defining interfaces to discover, to create, to publish and to access Grid services. OGSA does not address on its own any security mechanism such as authentication or secure service invocations. Instead, it relies on the security of the Web services.

2.1.5. Grid supporting tools

We describe in this section two important toolkits that support Grid computing at present: Globus and Legion. The development of these toolkits has strongly reflected the tendency of Grid computing.

2.1.5.1. Globus Toolkit

The Globus toolkit is one of the most important tools for Grid computing at present. It is the result of a joint project between University of Southern California, Argonne National Laboratory and The Aerospace Corporation started in 1997. Globus Toolkit provides services to manage the computational Grid (software and hardware) for distributed, high-throughput super-computing. The first birth version 1.0 of the toolkit in 1998 was deployed on the GUSTO testbed which involved more than 70 universities and institutes over the world in 1999. In 2000, more than 125 institutes over 3 continents joined the GUSTO. Version 2 of the toolkit, released in 2002, marked an important point in the first wave of Grid development where basic Grid services have been identified and tested. Version 3 of the toolkit (2003) starts the second wave of the Grid evolution focusing on the inter-operability and the integration of distributed services. Growing rapidly, Globus has become a powerful grid-enabled toolkit and is considered as a reference implementation of Grid components.

The toolkit comprises a set of basic services for the Grid's security, resource location, resource management, information, remote data management, etc. The services are designed with the principle of an "hourglass": the neck of the hourglass provides a uniform interface to access various implementations of local services [29]. The developer uses this interface to develop high-level services for his own needs.

The up-coming of Web services recently has considerably changed the inter-operability of Globus services. From the Global Grid Forum, an Open Grid Service Architecture (OGSA) [30] using Web services technologies has been proposed. Service architectures used in the old Globus toolkit version 1 and 2 (GT1 and GT2) have been rewritten to use OGSA (Globus Toolkit version 3- GT3). OGSA does not only provide a uniform way to access Grid services but it also defines the conventions in which new Grid services can be described (based on Web Service Description Language-WSDL) and integrated into the existing Grid system.

2.1.5.2. Legion toolkit

Legion is another toolkit for Grid computing. The first public release was made at Supercomputing '97 in San Jose, California, on November, 1997. In 2000, the Grid Portal for Legion has been in operation on npacinet- a worldwide grid managed by Legion on NPACI (the US National Partnership for Advanced Computational Infrastructure) resources.

Legion [39, 40], developed by University of Virginia also provides similar services as Globus but follows an object-oriented approach. From the Legion point of view, everything inside

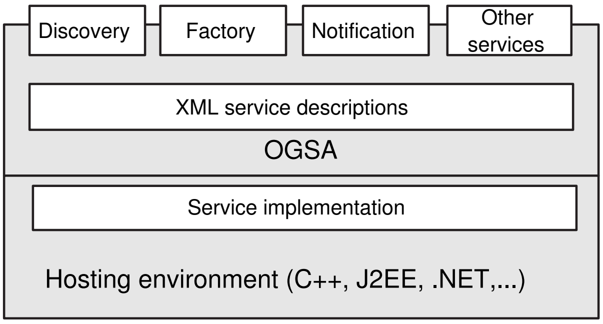


Figure 2.1: Service architecture in GT3: OGSA defines the service semantics, the standard interfaces and the binding protocol that is independent of the programming model that implements the service in the hosting environment

the environment, from a resource, a service to a running process, is an object. Legion defines a protocol and a message format for remote method invocation.

Legion contains a set of core objects. Each core object defines a specific functionality in the distributed system. Host object, for instance, is responsible for managing a resource such as making resource reservation or executing other objects on the resource. The user-defined object is based on the core objects to access the system. Between the core objects and the user objects there are object-object services which improve the performance of the system. The cache object, for example, is used to reduce the loading time of a user object from a persistent storage.

In the Legion object model, Class objects, differ from traditional object oriented models, are themselves active entities that play the role of the object containers. These containers are responsible for managing and placing objects instances on remote resources.

2.2. Programming models

Programming models are directly related to the application development. They define the way to describe the parallelism, the problem decomposition, the interactions, etc. Programming models cannot live apart from the environment. To exploit the power of a computational environment, programming models have to be carefully designed. The literature shows that currently there is no specific programming model specially designed for the Grid. Most models used on the Grid nowadays come from those used in the traditional parallel and distributed environments. Therefore we will focus on the distributed computing models and how suitably can we use them for the Grid.

Distributed computing has a quite long history of development of over 20 years. Many models have been investigated. We present in this section four important styles of parallel programming: the message passing, the distributed shared memory, the bulk synchronous parallel and the object-oriented approach.

2.2.1. Message passing model

Message passing is one of the most widely used models for parallel distributed programming. The model consists of tasks (or processes) running in parallel. The communication between tasks is explicitly specified by the programmer via some well-defined send and receive primitives. The message passing model provides programmers with a very flexible generic mean to develop parallel application. It can also deal well with the heterogeneity of the environment. However, message passing is a quite low-level programming model in which programmers have to manage all communication and synchronization among tasks.

Two well-known message passing tools up-to-date are the parallel virtual machine (PVM) [34] and the message passing interface (MPI) [42]. PVM was first developed in 1989 at Oak Ridge National Laboratory to construct a virtual machine that consists of network nodes. PVM allows the user to dynamically start or stop a task, add or delete a host to or from the virtual machine, send and receive data between two arbitrary tasks. On the Grid, PVM has two disadvantages. First, PVM does not provide any mean to manage the task binary codes. It is up to the programmer to specify the correct executable file and the corresponding hardware architecture, and to ship the codes to the proper place on the target PVM host. This considerably limits the flexibility in exploiting the performance from heterogeneous environments. Secondly, PVM does not provide any mean for resource discovery and users have to add/delete hosts manually to the system. The two disadvantages limit the scalability of the system as the number of nodes constituting the virtual machine grows.

MPI standard was born in April, 1993 with the first specification. MPI defines both the semantics and the syntax for the core message passing primitives that could be suitable for a wide range of distributed high performance applications. MPI is not a tool. It does not specify any information about the implementation of these primitives. Each vendor can provide his own implementation of the primitives that best fits his hardware architecture. Since MPI intends to just provide a common interface for message passing routines, it does not include any specification on process management, input/output controls, machine configuration, etc. All of these necessities depend on the vendor of the tool. The main advantage of MPI is the portability of MPI applications to various architectures. Nowadays, MPI-based tools and libraries have been the dominant factors in high performance computing.

Along with the rapid development of Grid computing and Grid infrastructures, some existing tools have been successfully ported to the Grid environment. MPICH-G [27, 50], a Globus [28]-based version of MPICH has been developed, allowing the current MPI applications to run on the Grid without any modification. The heterogeneity of the Grid can considerably affect the performance of MPICH-G if the tasks are not carefully placed. The quality of services has been taken into account in MPICH-GQ [69]. PVM and MPI have also been implemented on the Legion toolkit [40] via the emulation of the libraries to use the underlying Legion run-time library. Porting existing libraries to the Grid preserves users from rewriting the whole applications from scratch, so that existing applications only need to be recompiled to run on the Grid.

2.2.2. Distributed shared memory

Shared memory is an attractive programming model for designing parallel and distributed applications. Many algorithms have been designed based on the shared memory model. In the past, shared memory models were quite popular on massive parallel processing systems with the physical support of memory architectures. Following the amazing development of the networking technologies and the advances on microprocessors, high performance computing has a bias toward distributed processing with clusters, network of workstations, etc. To make use of exiting algorithms and applications on the distributed environment, an abstraction of shared memory on physically distributed machines has been built. This abstraction is known as Distributed shared memory (DSM).

Although DSM offers the programmer to freely use standard programming methods that exist on traditional multi-processor systems such as multi-threading or parallel loops but DSM usually results in poor performance and limits the scalability of applications compared to other distributed models such as message passing [14]. The DSM-based applications often work better if the programmer can specify the layout of memory and customize the memory access scheme.

Many DSM systems have been reported in the literature [61]. Some of the well-known ones are Munin [13], DiSOM [63] and InterWeave [76]. Munin is a software DSM system that implements the shared memory by some special annotations of access patterns on shared variables (e.g. read-mostly, write-once, write-many, etc.). Munin manages the memory consistency by choosing a suitable consistency protocol based on the access pattern. To reduce the communication overhead, Munin provides the release-consistent memory access interface [35] in which the memory consistency is only required at specific synchronization points. One big disadvantage of Munin is that it lacks heterogeneous support, a fundamental characteristic of the Grid. DiSOM is a distributed shared object memory system. Shared data items in DiSOM are represented as objects with type information. This information is used to deal with the heterogeneity of the environment. The memory consistency model in DiSOM is entry consistency [59] in which each data item has a synchronization variable and all access on that item will be quoted by the acquire/release operations on its corresponding synchronization variable. InterWeave model assumes a distributed collection of clients-the ones that use shared memory and servers-the ones that supply shared memory. Shared memory is organized as strongly typed blocks within a segment and is referred via the machine-independent pointer which consists of the host name, the path, the block name and the optional offset within that block. Interweave allows to access the shared memory as if it is local memory by trapping the signal upon a page fault. To reduce the communication overhead, InterWeave dates the shared data, tracks changes on the data and transmits only the changed parts to the client upon requested. InterWeave supports the heterogeneity by converting data into wire format before the transmission. One disadvantage of InterWeave is that it does not provide any mean for remote process creation. Hence, Interweave should be combined with other distributed tools to form a complete development environment for distributed applications.

Although DSM can facilitate the development of distributed applications. Its main disadvantage is the performance. Many issues, especially the granularity of shared data, the location of shared data and the heterogeneity support still need to be solved in order for the DSM model to be efficiently used on the Grid.

2.2.3. Bulk synchronous parallel

Bulk Synchronous Parallel (BSP) was proposed by L.G. Valiant in 1990 [82]. The BSP computation is defined as a set of components that perform some application tasks and a router that routes the point-to-point messages between pairs of components. The computation consists of a sequence of supersteps. Each superstep comprises three separate phases: first, all or a subset of components simultaneously does the computation on their local data; secondly, each component exchanges its data with other components (communication); and finally, all components are synchronized before moving to the next superstep (synchronization).

The separation of computation, communication and synchronization makes BSP a generic model that is clear and easy to manage. BSP is efficiently applicable on various kinds of architectures from shared memory multiprocessors to distributed memory systems. It offers a general framework to develop scalable and portable parallel applications. While the mixed communication-computation in other models such as in PVM, MPI makes it hard to predict the application performance, the separation of computation-communication gives the BSP model several advantages: the performance and the program correctness are easier to predict; the deadlock does not occur in a BSP program. However the disadvantages of BSP are: the different sizes of tasks can decline the possibility of overlapping between computation and communication; the overhead for synchronization is big; and the mapping between subproblems of a decomposition into sequence of components/supersteps is not obvious.

Since BSP was born, number of BSP tools has been developed. BSPlib [46] provides a defacto standard implementation of the BSP communication library. BSPlib consists of about 20 primitives that manage all communication between components. Two communication models supported in BSPlib are: direct remote memory access (DRMA) and bulk synchronous message passing (BSMP). In DRMA, a component (process) will explicitly register a local memory to the BSP system so that other components can put/get data to/from this memory remotely. In BSMP, each component explicitly uses the send/receive primitives to send or receive messages to/from other components.

ParCel-2 [11, 10, 52] developed at LITH/EPFL extends the BSP model in several ways.

First, ParCel-2 is a cellular programming language which allows the user to express the computation in cells. Several cells can be grouped together to form a bigger cell. Secondly the communication between cells has been typed with some specifications. Finally, ParCel-2 allows the synchronization to be performed after an integer multiple of the global superstep counter.

Heterogeneous Bulk Synchronous Parallel (HBSP) [86] extends the BSP model for heterogeneous computing by incorporating parameters that reflect the relative speeds of components. These parameters are used as the guideline for choosing a suitable size of work units for each component. BSP-G [79] expands BSPlib to the Grid by using the Grid services of the Globus toolkit for authenticating, executing BSP components. BSP-G provides an interesting portal of BSP application to the Grid environment although it does not solve the heterogeneity issue of both the Grid and the BSP components.

2.2.4. Object-oriented models

The object oriented approach is a promising solution to manage the complexity of developing HPC applications. While the object-oriented method has become a revolutionary concept that changes the rules in computer software engineering, in the domain parallel and distributed processing, the main use of object oriented techniques is focused on distributed client-server applications with some standards such as the Common Object Request Broker Architecture (CORBA) [4], Remote Method Invocation (RMI) [75] or Distributed Component Object Model (DCOM) [58]. The limitations of these standards are on the scalability and non-HPC design. There are also efforts to port non-object tools such as PVM, MPI to object oriented languages: JavaPVM [77], MPJ [12] but they are just the wrapper classes of the available functions and procedures. We will not consider such tools as following the object-oriented approach.

From the view of object activity, distributed object-oriented models can be categorized into two types: active objects and passive objects [19]. Active objects are resulted in the integration of processes and objects. Each active object possesses one or more processes that handle all object activities such as the acceptance of method invocations, synchronization, etc. When an active object is destroyed, all processes bound to this object are also terminated. Active objects are natural and simple in distributed systems.

Passive objects, on the other hand, are separated completely from the process. A single process can be used to execute several passive objects during its life time. The advantage of passive object model is that there is no limit number of processes bound to an object. However, it may be difficult and expensive to map objects to processes in distributed environments where the objects does not usually share the same memory address space.

There are number of researches on parallel and distributed object systems. They focus on two directions: developing object-oriented languages and constructing supporting libraries-tools for existing systems.

2.2.4.1. Language approach

On the language approach, Orca [3], MPL [41], PO [22, 21] and Synchronous C++ (sC++) [64] are some examples. Orca provides a new language based on shared data objects in distributed environments. The programmer has to explicitly create processes. Orca objects are passive objects that can be passed from one process to another process upon process creation. Shared data inside objects can be manipulated from processes via high-level object interfaces. Some important properties of object-oriented programming such as inheritance, polymorphism are not explicitly supported.

MPL is an extension of C++ with some so-called *metat* classes for parallel execution. MPL follows the active-object data-driven model. The parallelism is achieved by concurrent invocations on these objects. The Mentat runtime system is responsible for instantiating *mentat* objects, invoking methods and keeping objects consistency. The *metat* object supports only asynchronous invocation and is not shareable.

PO also follows the active object model with the capability of deciding when and which invocation requests to serve. Inside each PO object, a parallel part is responsible for interfacing between the methods and the outside world. Method invocations are carried out by using one of three communication modes: synchronous, asynchronous and future mode. In the asynchronous mode, the client is not blocked for the results of the invocation. The synchronous mode blocks the client until the method execution returns. The future mode is a non-blocking mode in which the client provides a "call back" address to which the server will store the return values of the invocation. One innovation of PO is the ability to specify the high-level directives for the object allocation for each PO class through the Abstract Configuration Language (ACL). The run-time system will use these directives to choose a suitable resource for a PO object.

Synchronous C++ (sC++) is yet another object oriented programming language that follows the active object model. Synchronous C++ extends C++ to distributed environments by adding a special part to each object class called the class *body*. In each sC++ object, the body is executed on the control thread of the object. It is responsible for scheduling methods that are ready to be invoked. Any method invocation can only occurs when the corresponding body explicitly accepts the method (server side) and the client makes a call to that method. The sC++ body part of the object provides a flexible way for checking the constraints and the integrity of methods. However, the execution in each sC++ object is atomic which limits the ability to achieve the intra-object parallelism.

2.2.4.2. Supporting tool approach

COBRA [65] and Parallel Data CORBA [51] extend the CORBA standard by encapsulating several distributed components (object parts) within an object and by implementing the data parallelism based on data partitioning. Data input on an object will be automatically split and distributed to several object parts that can reside in different memory address spaces. The user can access high performance computing services provided by these tools as if they accessed standard CORBA objects. Both COBRA and Parallel Data CORBA concentrate on interfacing parallel computation services with the outside world, rather than focusing on the parallel elements of the application.

HPC++ [49] is a C++ library and language extension of C++ for portable and distributed C++ programming. The HPC++ library consists of primitives to register methods, to pack or unpack data and to invoke remotely registered methods. HPC++ is a quite low level library that should be used with other tools to facilitate the manipulation of objects.

2.3. Requirements for high performance Grid applications

Along with the rapid development of the Grid and distributed computing, one main question has emerged: *How to exploit the performance from the highly distributed heterogeneous environment?* Clearly, the answer should come from both the infrastructure and the application structure.

2.3.1. New vision: from resource-centric to service-centric

The computational Grid makes the traditional assumption of performance as the number of processors involved in the computation become obsolete due to the heterogeneity of resources. The traditional *resource-centric approach* in which the user requests to run the application on some explicitly specified resources has become hardly feasible on the Grid environment due to the large number of dynamic resources. New issues of the Grid lead to the quest for a new method for executing and developing applications. The *service-centric approach* is to answer this quest. The application following the service-centric approach will not ask for the resources but for the services. It will ask the infrastructure to obtain necessary services as the abstractions of functions regardless the service location. The infrastructure then performs the service discovery to find a suitable service, to authenticate the service and to grant the access of the service to the user.

Services are usually developed by system developers that hide the complexity of the environment from the user by allowing the user to access high-level functionalities of the environment. All details of the implementation are encapsulated inside the services. By this way, the application programmer can focus on the implementation parts of the problem domain.

2.3.2. Application adaptation

In addition to the change of the infrastructure from the resource-centric to service-centric approach, programming models for the Grid also need to be further investigated.

The literature shows that programming models depend very much on the execution environment of the applications. Each programming model is usually fit to a specific environment. The shared memory model, for example, is suitable for SMP systems or distributed systems with very high speed inter-network connections while the message passing model is widely used in rather distributed environment such as clusters or networks of workstations with slower communication.

Programming models on the Grid should be able to deal with the Grid issues such as the heterogeneity, the communication latency, the dynamic and instability, etc. The application needs to adapt itself to the environment. The adaptation can be:

- Dynamic task sizes. The size of a task should be parameterized. Each task has different requirements on the resource. In other words, we use the heterogeneity of application components to deal with the heterogeneity of the environment.

- Different level of parallelism. Each application consists of several configurations. Each configuration represents a level of parallelism. Depending on the availability of resources at run-time, a suitable configuration will be executed. This is crucial to real-time applications on the Grid since the dynamics and volatility of the Grid oppose the fixed run-time configuration of the application.

- Dynamic utilization of resources. The resource will be assigned to the application on demand and the application should not occupy resources if it does not really need them. When a component completes its task, the resource should be released.

- Active reaction to failures. The application should be able to detect failures of components and to replace the failed component by a new one on a different suitable resource.

To allow the adaptation, high performance Grid applications should somehow be able to describe the requirements of distributed components and to use the infrastructure services, according to the service-centric approach, to discover the suitable resources and to execute the components on those resources.

2.4. Summary

We have surveyed the state-of-the-art of the Grid computing at two different levels: the infrastructure development and the programming model. There are still many challenges need to be overcome. In our opinion, one of the key challenges is the efficient exploitation of the seamless power of the Grid for high performance computing applications. This challenge needs to be addressed at both levels. The traditional assumption of performance as the number of processes has become obsolete due to the highly heterogeneous resources. Traditional scheduling algorithms seem not to be suitable to the Grid due to the unpredictable and volatile properties of the environment. Traditional programming models pose many difficulties and limitations to be efficiently used on the Grid.

To extract the high performance of the Grid for applications, application adaptation is required. Such adaptation is addressed in different ways: different task sizes, different grain of parallelism, dynamic resource utilization and active reaction to failures.

Throughout the dissertation, we will focus on the main challenge of how to efficiently achieve the power of the Grid for high performance applications and particularly applications with time constraints through the application adaptation. The state-of-the-art of Grid computing shows that at the moment, there is no metric to measure the efficiency of the Grid application. The old definition of efficiency as the ratio between the speedup over the total number of processors is not suitable in this context due to the heterogeneity. We will not develop a new metric to measure the efficiency but we will consider the efficiency in our sense as the maximum amount of speedup that an application can gain from the Grid environment and the ability of an application to satisfy the user time requirements.

We study the adaptation from two different points: from the level of infrastructure to the programming language and programming paradigm to the conceptual level of parallelization.

Around the main endeavor that we address in the thesis, we also cover some related issues of the Grid such as resource management and fault tolerance. Although we will not study other issues like resource connectivity, security, information safety, etc. but we still count them as important problems of the Grid.

Chapter 3

Parallel object model

3.1. Introduction

Object-oriented methods provide high level abstractions for software engineering. The nature of objects shows many possibilities of parallelism: a) the parallelism among a collection of objects where each object may live independently from others; b) the parallelism inside each object: some operations on the same object can occur concurrently. In distributed environments such as the Grid, having all objects running remotely usually is not efficient due to the communication bottle-neck problem. Thus, we need to answer the two questions:

- Question 1: which objects will be remote objects?
- Question 2: where does each remote object live?

The answers, of course, depend on what objects are doing and how they interact with each other and with the outside world. In other words, we need to know the communication-computation requirements of objects. The parallel object model that we present in this chapter provides an object-oriented approach for requirement-driven high performance applications in the distributed heterogeneous environment.

3.2. Parallel object model

We envision parallel objects as the generalization of the traditional object such as in C++. One important support for parallelism is the transparent creation of parallel objects by dynamic assignments of suitable resources to objects. Another support is various mechanisms of invocation concurrency: concurrent, sequential and mutex (see section 3.4).

In our model, a parallel object has all properties of traditional objects plus the following ones:

- Parallel objects are shareable. References to parallel objects can be passed to any method regardless wherever it is located (locally or remotely). This property is described in section 3.3.

- Syntactically, invocations on parallel objects are identical to invocations on traditional sequential objects. However, parallel objects support various method invocation semantics: synchronous, asynchronous, sequential, mutex and concurrent. These semantics are discussed in section 3.4.

- Objects can be located on remote resources and in a separate address space. Parallel objects allocations are transparent to the user. The object allocation is presented in section 3.5.

- Each parallel object has the ability to dynamically describe its resource requirement during its lifetime. This feature is discussed in detail in section 3.6.

It has to be mentioned that by default, the parallel object is in the inactive state. The object can only be activated upon executing a method invocation request. The waiting for and accepting incoming requests at the server side are performed implicitly and transparently to the user. Hence the user does not have to implement himself the object body to schedule the acceptance of method invocations. We believe by this way, users can simplify the control of the object execution, thus allowing a better integration into other software components.

3.3. Shareable parallel objects

All parallel objects are shareable. Shared objects with encapsulated data provide a means for users to implement global data sharing in distributed environments. Shared objects can be useful in many cases. For example, Fig. 3.1 illustrates a scenario of using shared objects: *Input* and *Output* objects are shareable among *Worker* objects. *Worker* gets work units from *Input* which is located on the data server, performs the computation and stores the results on the *Output* located at the user workstation. The results from different *Worker*s can be automatically synthesized and visualized inside *Output*.

In order to share a parallel object, our model allows parallel objects to be arbitrarily passed from one place to the other as arguments of method invocations. It is the run-time system, not the user, which is responsible for setting up the interface and managing the object references so that the object is only physically destroyed if there is no reference to the shared object.

One important issue of object sharing is data consistency. The parallel object model provides different method invocation semantics (section 3.4) to allow users to define the desired level of consistency.

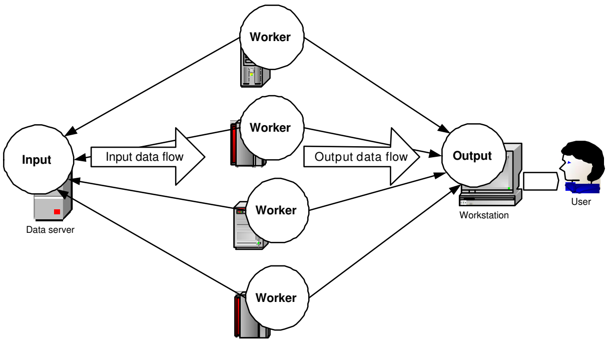


Figure 3.1: A usage scenario of shareable objects in the master-worker model

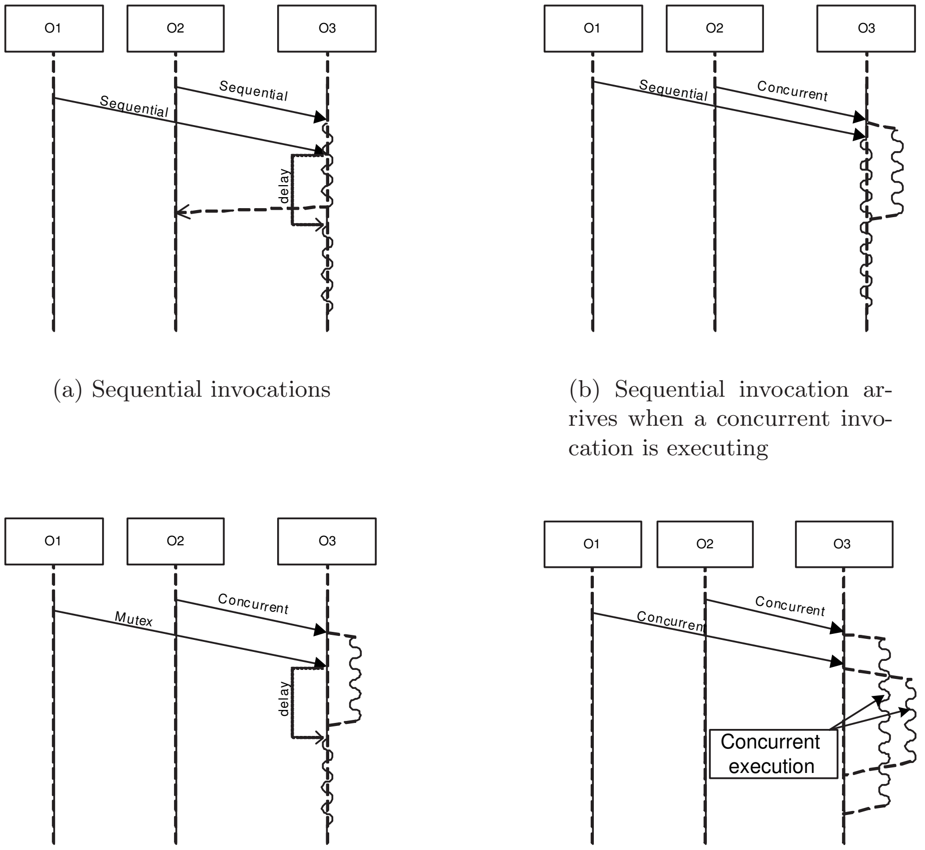
3.4. Invocation semantics

Syntactically, method invocations on parallel objects are identical to those on traditional sequential objects. However, each method in a parallel object is associated with different invocation semantics. These semantics are defined at both sides of the parallel object:

- Interface semantics-the semantics that affect the caller of method invocations:
  - *Synchronous invocation*: the caller waits until the execution of the requested method on the object side is finished and returned the results. This corresponds to the traditional method invocation.
  - *Asynchronous invocation*: the invocation returns immediately after sending the request to the remote object. Asynchronous invocation is important to exploit the parallelism because it enables the overlapping between computation and communication. However, at the time the execution returns, no computing result is available yet. This excludes the invocation from producing results. The results can be actively returned to the caller object if the callee knows the "call back" interface of the caller. This feature is well supported in our parallel object model by the fact that the interface of a parallel object can be passed as an argument to other parallel objects during the method invocation (the call back object interface).

- Object-side semantics-execution semantics of methods inside each parallel object:
  - *Sequential invocation*: the method is executed sequentially, i.e. when several other parallel objects invoke simultaneously sequential methods on one parallel object, these requests will be served sequentially (Fig. 3.2(a)). Nevertheless, other concurrent methods that have been previously started can still continue their normal works (Fig. 3.2(b)). The executions of sequential methods guarantee the serializable consistency of all sequential methods in the same object.

- *Mutex invocation*: this is the most restricted form of method invocation that guarantees the atomic execution of the method within a parallel object. The request is executed only if no other instance of methods is running. Otherwise, the current method will be blocked until all the others (including concurrent methods) are terminated (Fig. 3.2(c)). Mutex invocations are important to synchronize concurrencies and to assure the correctness of shared data state inside the parallel object (e.g. to implement mutual exclusive write on the same data).
- *Concurrent invocation*: the execution of the method occurs in a new process (thread) if no sequential or mutex invocation is currently invoked (Fig. 3.2(d)). All invocation instances of the same object share the same object data attributes. The concurrent invocation is important to achieve the parallelism inside each parallel object and to improve the overlapping between computation and communication.



(a) Sequential invocations

(b) Sequential invocation arrives when a concurrent invocation is executing

(c) Mutex invocation is delayed until all concurrent invocations are terminated

(d) Concurrent invocations

Figure 3.2: Object-side invocation semantics when several other objects (O1, O2) invoke a method on the same object (O3)

All invocation semantics are specified by the programmer at the design phase of parallel objects.

3.5. Parallel object allocation

To achieve the goal of dynamic utilization of computational resources and the ability to adapt to the changes from both the environment and the user, an object system should be able to dynamically create and destroy objects. In our parallel object model, the creation of parallel objects is driven by the high-level requirements on the resource where the object runs (see section 3.6). The user only needs to describe these requirements. The allocation of parallel object is then transparent to users and should be managed by the run-time system. The allocation process consists of three phases: first, the system finds a resource where the object will live; then the object code is transmitted and executed on that resource; and finally, the corresponding interface is created and connected to the object.

3.6. Requirement-driven parallel objects

Along with changes in parallel and distributed processing toward web and global computing, there is a challenging question of how to exploit high performance in highly heterogeneous and dynamic environments. We believe that for such environments, the high performance can only be obtained if the two following conditions are satisfied:

- The application should be able to adapt to the environment.
- The programming environment should somehow enable application components to describe their resource requirements.

The application adaptation to the environment can be fulfilled by multi-level parallelism, dynamic utilization of resources or adaptive task size partitioning. One solution is to dynamically create parallel objects on demand that will be expressed in section 5.1 of chapter 5 where we describe the ParoC++.

Resource requirements can be expressed in the form of quality of services that components require from the environment. Number of researches on the quality of service (QoS) has been performed [32, 47, 36]. Most of them focus on some low-level services such as network bandwidth reservation, real-time scheduling, etc.

Our approach integrates the user requirements into parallel objects in the form of high-level resource descriptions. Each parallel object is associated with an *object description* (OD) that depicts the characteristics of resource used to execute the object. The resource requirements in OD are expressed in terms of:

- Resource name (host name) (low level description, mainly used to develop system services).
- The maximum computing power that the object needs (e.g. the number of MFlops needed).
- The maximum amount of memory that the parallel object consumes.
- The communication bandwidth/latency with its interfaces.

An OD can contain several items. Each item corresponds to a type of characteristics of the desired resource. The item is classified into two types: strict item and non-strict item. Strict item means that the designated requirement must be fully satisfied. If no satisfying resource is available, the allocation of parallel object fails. Non-strict item, on the other hand, gives the system more freedom in selecting the resource. A resource that partially matches the requirement is acceptable although a full qualification resource is the preferable one. For example, the following OD:

```
"power= 150 MFlops ?:  100MFlops; memory=128MB"
```

means that the object requires a preferred performance 150MFlops although 100MFlops is acceptable (non-strict item) and a memory storage of at least 128MB (strict item).

The construction of OD occurs during the parallel object creation. The user will provide an OD for each object constructor. The OD can be parameterized by the input parameters of the constructor. This OD is then used by the runtime system to select an appropriate resource for the object.

It can occur that, due to some changes on the object data or some increase of the computation demand, the OD needs to be re-adjusted during the life time of the parallel object. If the new requirement exceeds some threshold, the adjustment may invoke the object migration. Object migration consists of three steps: first, allocating a new object of the same type with the current OD, then, transferring the current object data to new object (assignment) and finally, redirecting and re-establishing the communication from the current object to the newly allocated objects. The migration process should be handled by the system and be transparent to the user. The current implementation of the parallel object model, which we will describe in chapter 5, does not support the object migration yet.

3.7. Summary

Adaptive utilization of the highly heterogeneous computational environment for high performance computing is a difficult goal. The adaptation consists of two forms: either the application components should somehow decompose dynamically, based on the available resources of the environment, or the components allow the infrastructure to select suitable resources by providing descriptive information about the resource requirement.

We have addressed these two forms of adaptation by introducing the parallel object model: dynamic parallel object creation and deletion; and requirement-driven object allocation. Parallel object is a generalization of traditional sequential object model with the integration of user requirements via object-description into the shareable object. Although parallel objects are distributed, they clear the resource boundary of the distributed environments inside the application by the ability to be arbitrarily passed from one place to the others inside the application transparently via method invocations. The parallelism can be achieved by concurrent operations inside each parallel object (intra-object parallelism) as well as simultaneous operations among objects (inter-object parallelism).

In chapter 5, we will present an implementation of our parallel object model in an object-oriented programming system called ParoC++.

Chapter 4

Parallelization scheme

4.1. Introduction

Many practical problems require that the execution should be completed within a user-specified amount of time. We refer such problems as *time constrained problems* or *problems with time constraints*. Real-time applications are a special kind of these time constrained problems.

Number of on-going researches on time constrained problems focus on various aspects of scheduling issues such as in real-time CORBA [62], heterogeneous task mapping [9, 57] or multiple variant programming methodology. Multiple variant programming, for instance, enables the user to elaborate a number of versions to solve the problem into a single program. Each version has a different level of computational requirements. Depending on the environment, a suitable version will be automatically selected for execution. In [48], the author describes an evolution approach for scheduling several variances of independent tasks on a set of identical processors to minimize the total violations of deadline. Gunnels, in [44], presents variances of matrix multiplication algorithms and the evaluation of required performance based on the shape of matrices.

In this chapter, We present an original approach for solving time constrained problems based on the *dynamic parallelism*. Dynamic parallelism enables applications to exploit automatically and dynamically a suitable grain of parallelism, depending on the available resources. This is an important issue to efficiently exploit the computing power of the Grid since the applications should adapt themselves to the heterogeneity of resources inside the environment.

4.2. Parallelization scheme

We introduce the notion of *parallelization scheme* that allows expressing potential parallelism and time constraints for a given problem. A "*problem*" in this context means a program that

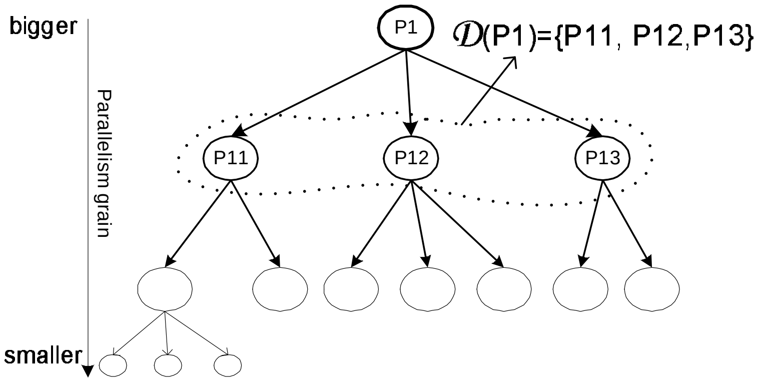


Figure 4.1: Decomposition Tree

the user needs to execute. The process of executing the problem to produce the outcome is called a *solution to the problem.*

Definition 4.1 (Parallelization scheme). A *parallelization scheme* consists of a decomposition tree (DT) defining how to decompose the problem at different levels and a set of decomposition dependency graphs (DDG), one for each non-leaf node of DT, defining the partial order of executions of sub-problems within each decomposition. If P is the original problem to solve, then the parallelization scheme of P is denoted:

$\mathcal{S}(P) =< DT(P), \{DDG(P_i)|P_i \in DT(P)\} >$

The DT and the DDG are defined bellow.

Definition 4.2 (Decomposition tree). If we can replace the solution of a problem P_i by the solution of the set of problems $\{P_{i1}, P_{i2}, \dots P_{in}\}$ then we denote this set as $\mathcal{D}(P_i) = \{P_{i1}, P_{i2}, \dots P_{in}\}$ and we call it the *decomposition set* of P_i. The process of deriving $\mathcal{D}(P_i)$ from P_i is called a *decomposition step.*

The *decomposition tree* of a problem P_i, denoted $DT(P_i)$, is constructed by recursively applying decomposition steps to each element of the decomposition set until no more decomposition step is possible.

The decomposition tree DT(P) represents one possible way to decompose a given problem P at all different levels with the following properties:

- The relationship between P and $\mathcal{D}(P)$: a solution can be obtained by solving P alone *or* by solving $\mathcal{D}(P)$.
- The relationship among problems within the same decomposition set $\mathcal{D}(P)$. Consider $\mathcal{D}(P) = \{P_1, P_2 \dots P_n\}$, solving $\mathcal{D}(P)$ means solving all P_1 *and* P_2 *and* P_3 *and* ... *and* P_n. Here we do not take into account yet the dependencies between problems $P_i \in \mathcal{D}(P)$.

Definition 4.3 (Decomposition Dependency Graph). Consider the decomposition set $\mathcal{D}(P)$ of a problem P. The *decomposition dependency graph* of P is defined as a directed acyclic graph DDG(P) $=\langle \mathcal{D}(P), E\rangle$ with the set of vertices $\mathcal{D}(P)$ and the set of edges $E \subseteq \mathcal{D}(P) \times \mathcal{D}(P)$. Each edge $e = \langle P_i, P_j\rangle \in E$ means P_j should be executed only after P_i has been solved.

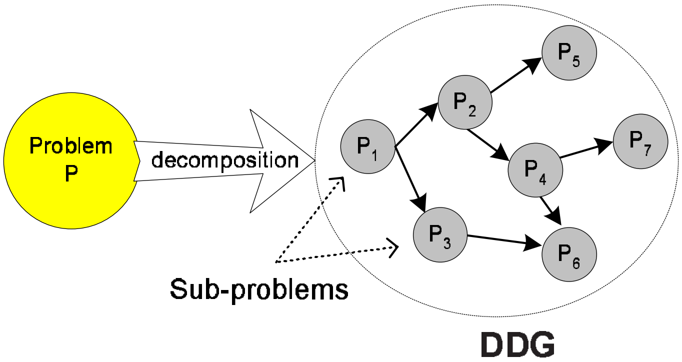


Figure 4.2: Decomposition Dependency Graph

$DDG(P_i)$ represents the partial order in which the set of sub-problems $\mathcal{D}(P_i)$ must be solved in order to solve P_i. While the decomposition tree gives an overall view of the parallelization process, the DDG expresses the sequential constraints of sub-problems within a decomposition step. DDG is similar to the data flow graph however, DDG is not a data flow graph because the execution of two sub-problems connected by an edge in DDG must be in sequential order: one must be completed before the other can start. For instance, two pipelined sub-problems form an edge in the data flow graph but not an edge in DDG because these two pipelined sub-problems are executed simultaneously, not in strictly sequential order. Figure 4.2 shows a decomposition step; the original problem is decomposed into 7 sub-problems. The graph on the right side illustrates a possible DDG of the original problem: sub-problem 1 should complete before sub-problems 2 and 3 can start and so on.

Definition 4.4 (Decomposition cut). A *decomposition cut* of a tree is a sub-set χ of nodes of the *decomposition tree* having the following property: for every path from the root to any leaf, the set ζ of nodes on this path has the following property: $\mid \zeta \cap \chi \mid = 1$.

Any path from the root to a leaf cuts each decomposition cut at exact one point. Figure 4.3 illustrates several decomposition cuts of a decomposition tree. In the figure, the sets $\{B, C, G, H, I\}$ and $\{E, F, C, D\}$ are two decomposition cuts. The set $\{E, F, G, H, I\}$ is not a decomposition cut since it does not cut the path $A - C$. The set $\{E, F, C, D, G, H, I\}$ is not either because it cuts the path $A - D - H$ at two points (D and H).

Theorem 4.5. Each decomposition cut of the decomposition tree is a solution to the problem.

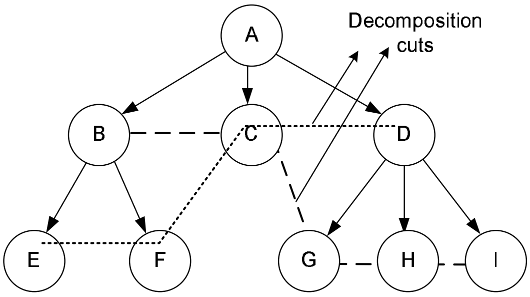


Figure 4.3: Decomposition cuts

Proof:

We first define the predicate *solve* on a problem and a set of problem as following: if P is a problem then $solve(P)$ means executing the problem P on a computing system. Let $\mathcal{S}$ be a set of problems, $solve(\mathcal{S})$ means solving all problems in $\mathcal{S}$:

$solve(\mathcal{S}) \Leftrightarrow \forall P_i \in \mathcal{S}, solve(P_i)$

Let χ be a decomposition cut, we construct a finite series of decomposition cuts $\chi_1, \chi_2, \ldots, \chi_m$ as following:

$\chi_1 = \chi$

Let:

$\chi_i^L = \{P_k : P_k \in \chi_i \text{ and } P_k \text{ is a leaf}\}$: the sub-set of leaf nodes of χ_i,

$\chi_i^N = \chi_i - \chi_i^L$: the sub-set of non-leaf nodes of χ_i and

$\mathcal{D}^*(\chi_i^N) = \bigcup_{P_k \in \chi_i^N} \mathcal{D}(P_k)$: the union of all decomposition sets of non-leaf nodes of χ^i.

The next set χ_{i+1} in the series is constructed from χ_i as:

$$\chi_{i+1} = \chi_i^L \cup \mathcal{D}^*(\chi_i^N) \tag{4.1}$$

The series is constructed by increasing i and applying equation 4.1 until χ_{i+1} does not contain any non-leaf node ($\chi_{i+1}^L = \emptyset$).

In other words, we replace each non-leaf node in χ_i by its decomposition set to produce χ_{i+1}.

If χ_i is a decomposition cut, for any path from the root to a leaf of DT(P), let ζ be the set of nodes on this path:

$\mid \zeta \cap \chi_i \mid= 1$

$\Rightarrow\mid \zeta \cap (\chi_i^N \cup \chi_i^L) \mid= 1$

$\Rightarrow\mid (\zeta \cap \chi_i^N) \cup (\zeta \cap \chi_i^L) \mid= 1$

$\Rightarrow\mid (\zeta \cap \mathcal{D}^*(\chi_i^N)) \cup (\zeta \cap \chi_i^L) \mid= 1$ (non-circle property of the tree)

$\Rightarrow\mid \zeta \cap (\mathcal{D}^*(\chi_i^N) \cup \chi_i^L) \mid= 1$

$\Rightarrow\mid \zeta \cap \chi_{i+1} \mid= 1$

Hence χ_{i+1} is also a decomposition cut.

From the definition of the decomposition tree, if P_i is a non-leaf node of DT(P). We have:

$$solve(P_i) \Leftrightarrow solve(\mathcal{D}(P_i))$$
$$solve(\chi_{i+1}) \Leftrightarrow solve(\chi_i^L \cup \mathcal{D}^*(\chi_i^N))$$
$$\Rightarrow solve(\chi_{i+1}) \Leftrightarrow solve(\chi_i^L) \text{ and } solve(\mathcal{D}^*(\chi_i^N))$$
$$\Rightarrow solve(\chi_{i+1}) \Leftrightarrow solve(\chi_i^L) \text{ and } solve(\chi_i^N)$$
$$\Rightarrow solve(\chi_{i+1}) \Leftrightarrow solve(\chi_i^L \cup \chi_i^N)$$
$$\Rightarrow solve(\chi_{i+1}) \Leftrightarrow solve(\chi_i)$$
$$\Rightarrow solve(\chi) \Leftrightarrow solve(\chi_m)$$

Due to the way we construct the series, the decomposition cut χ_m contain all leaf nodes of DT(P). From the construction of the DT, it is obvious that all leaf nodes form a solution to the problem. Therefore χ is also a solution to the problem.

□

Now we will evaluate the number of potential solutions that can be obtained from the decomposition tree. We will consider a special case where each decomposition step of a problem produces the same number of sub-problems. As we know that each decomposition cut is also a solution to the original problem, the following definition and theorem give estimation about the number of potential solutions.

Definition 4.6 (N-complete decomposition tree of degree δ). *An N-complete decomposition tree of degree* δ, denoted $\mathcal{T}(\delta, N)$ is a B-tree [7] of degree δ whose height is N and each node except the leaf nodes has exactly δ child nodes.

Theorem 4.7. The total number U_n of decomposition cuts of $\mathcal{T}(\delta, N)$ satisfies:
$U_n \geq 2^{\delta(n-1)}$ for $n \geq 1$)

Proof: We notice that $\mathcal{T}(\delta, N)$ is constructed by δ trees $\mathcal{T}(\delta, N-1)$. Therefore the number of cuts U_n can be calculated as the combination of all of cuts of δ trees $\mathcal{T}(\delta, N-1)$ plus the cut at the root: $U_n = (U_{n-1})^\delta + 1$. For $n \geq 1$, we can easily see that $U_n \geq 2^{\delta(n-1)}$. □

Solving a problem with the parallelization scheme is a multi-choice problem: the user has to choose among all potential solutions the *best* one. We choose the solving time as the criterion to consider for the best solution-the solution that runs fastest. The dynamics, the volatility and the heterogeneity of resources inside computational environments such as the Grid prevent the user from performing the selection in advance. Instead, the user must perform the selection at run-time. From theorem 4.7, we can deduce that an exhaustive search for the best solution is an NP-algorithm. Therefore, instead of finding the best solution, we will focus on finding an acceptable solution, i.e. the solution that is solved within a user specified amount of time. In the next section, we will present an algorithm to find such acceptable solutions based on a *parallelization scheme* that uses user estimates for the complexity inherent in (sub-)problems.

4.3. Solving time constrained problems

This section presents an approach for solving problems with time constraints on the Grid. The user will specify the amount of time within which he wants his problem to be solved and we need to find a solution that satisfies this time constraint. Given a time constraint, a solution that satisfies this time constraint if the total execution time of that solution is smaller than or equal the time constraint.

4.3.1. Problem statement

Given problem P with the time constraint T. We assume:

1. The parallelization scheme $\mathcal{S}(P)$ of P is known (we know how to construct it).

2. For each sub-problem P_i in $DT(P)$, the complexity $\mathcal{C}(P_i)$ of P_i is known (or can be estimated).

3. The set of available resources as well as their characteristics are not known in advance.

The first assumption requires the programmer to specify the parallelization scheme. The second assumption describes the type of problems (known complexity). In such problems, for a given input, we should be able to compute the total computing power needed (e.g. in terms of MFlops). In many cases, the exact number is not known, therefore, an estimation is acceptable.

The third assumption is about the computational environment. We tend to develop a model for some uncertainty environments such as Grid [31] or Peer-to-Peer [5]. In such environments, the user needs to discover resources on the fly. Based on the discovery results, the model should automatically apply a suitable grain of parallelism to execute the application.

We state our objective as follow: *given a problem P and its parallelization scheme $\mathcal{S}(P)$, solve P within the user specified time constraint T.*

We need to deal with how to find a suitable solution (among all potential solutions) that satisfies the time constraint. This is a lack information problem since we do not know about the resource characteristics. In addition, we only know the time constraint $T_0 = T$ of the root problem P_0 in $\mathcal{S}(P)$.

This is similar to the task scheduling problem in which the user needs to choose, among possible assignments, the one that satisfies some criteria. However, our problem is more complex since we have to find a suitable decomposition cut that fits the computational environment. Moreover, since the computational environment is dynamic and the resource discovery is only performed at run-time, the instance of the solution can vary during time.

4.3.2. Algorithm

Input:

- A decomposition tree whose root is P_0.
- The time constraint T_0.

Output: A solution that satisfies the time constraint.

Algorithm:

```
S1 Let $P = P_0$ (the root of the decomposition tree).
   Let $T = T_0$ (the time constraint provided by the user).
S2 Find a resource with the effective power $\mathcal{C}(P)/T$.
   If success, assign $P$ to that resource to solve sequentially and return.
   If not, go to step S3.
S3 If $\mathcal{D}(P_0) = \emptyset$ then return fail.
   For each child node $P_i$ of $\mathcal{D}(P)$:
     • Evaluate the time constraint $T_i$ of $P_i$ (see section 4.4).
     • Perform recursively the algorithm with inputs: the decomposition tree whose
       root is $P_i$ and the time constraint $T_i$
```

The algorithm shows how time constrained problems can be solved. We start with the root P_0 where we know the time constraint. From the assumption of knowing the complexity $\mathcal{C}(P_0)$, we can estimate the computing power of the resource needed. We first try to solve the problem at the root sequentially by allocating the resource in the environment(S1). If no such a resource exists, we need to find an alternative solution based on the composition set $\mathcal{D}(P_0)$. We know the time constraint T_0 to solve $\mathcal{D}(P_0)$. However, the time constraint for each problem $P_i \in \mathcal{D}(P_0)$ is unknown at the moment. If the $DDG(P_0)$ has no edge (all sub-problems are independent), all sub-problems can be solved in parallel. Hence the time constraint of sub-problems in this case is also the time constraint T_0 of the original problem P_0. Otherwise, edges in $DDG(P_0)$ infer the solving dependencies among sub-problems: some sub-problems must be solved before others. Because all sub-problems must be solved within the time T_0, an increase of the time constraint of a sub-problem may cause the decrease of the time constraint of other sub-problems. In other words, time constraints of sub-problems in this case are dependent. We provide a method to estimate these time constraints in section 4.4. When all time constraints of problems in $\mathcal{D}(P_0)$ are evaluated, we can repeat this process for the sub-trees whose roots are in $\mathcal{D}(P_0)$.

4.4. Time constraints in the decomposition tree

Let us consider a decomposition step of problem P into sub-problems $\mathcal{D}(P) = \{P_1, P_2, \ldots, P_n\}$. Suppose that we know the time constraint T of P. We need to find the time constraints $T_1, T_2, \ldots, T_n$ for $P_1, P_2, \ldots, P_n$.

First, we will build the *sequential diagram* from the decomposition dependency graph (DDG). Sequential diagram and DDG are two different presentations of sub-problem dependencies within a decomposition step. While the DDG directly shows the dependencies between two sub-problems, the sequential diagram represents the best start time of a sub-problem without taking into account of the resource requirements. Section 4.4.1 will present an algorithm to construct the sequential diagram from the decomposition dependency graph.

Definition 4.8 (Sequential diagram). Given a DDG(P) of problem P, a *step* is the minimum execution unit so that at least one sub-problem is solved. The *sequential diagram* is a directed acyclic graph $< \mathcal{V}, E >$ where $\mathcal{V} = \{S_1, S_2, \ldots, S_m\}$ is the set of steps and E is the set of $n =\mid \mathcal{D}(P) \mid$ edges whose labels are $P_1, P_2, \ldots, P_n$ such that:

- If P_x is the label of an edge $< S_i, S_j >$ (denoted: $< S_i, S_j > \doteq P_x$) then P_x must not start before the step S_i and P_k must complete before the step S_j.
- $\forall < S_i, S_j > \in E \Rightarrow i < j$: the start time cannot be after the finish time of the same sub-problem.
- $\forall i < m \Rightarrow < S_i, S_{i+1} > \in E$: between two consequent steps, there exists at least one problem to be solved.
- $\forall < S_i, S_j > \doteq P_x, < S_l, S_m > \doteq P_y \in E$, if $< P_x, P_y > \in DDG(P) \Rightarrow j \leq l$. That means if P_y depends on P_x, it can not be executed before P_x is solved.

Note that it is possible to have multiple edges between two steps.

Figure 4.4 shows an example of DDG and its sequential diagram. The sequential diagram specifies the start points and the end points of sub-problems. For example, sub-problem P_1 is executed in step S_1 and completed before step S_2. Sub-problems P_2 and P_3 are started in step S_2. P_2 must complete before step S_3 and P_3 must complete before step S_4 and so on. The sequential diagram is used to compute the time constraints of sub-problems and to schedule the tasks in order to satisfy the overall time constraint.

4.4.1. Algorithm to find the sequential diagram

Input A DDG of problem P.

Output A sequential graph G.

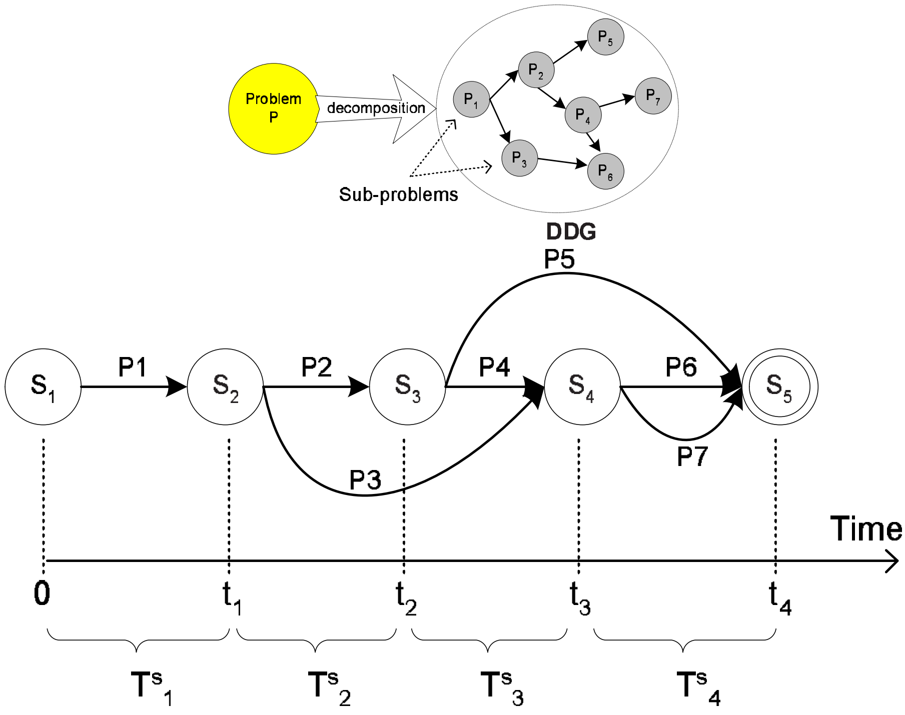


Figure 4.4: The decomposition dependency graph and its corresponding sequential diagram

Algorithm

S1 Let $L(P_i)$ be the mark status of vertex P_i in DDG(P).

Initially, $\forall P_i, L(P_i) = 0$ (”unmarked”).

S2 For every unmarked vertex P_i with no unmarked input vertex

- Let $s = 0$ if P_i has no ingoing edge,
  otherwise $s = max\{L(P_k) | < P_k, P_i > \text{ is an edge in DDG(P)}\}$.
- Mark P_i as $L(P_i) = s + 1$

Repeat this step until all vertices are marked.

S3 Let $m = max\{L(P_i)\}$. The sequential diagram consists of $(m + 1)$ vertices S_1, S_2, $\ldots, S_m$, $S_{m+1} = S_s$ and n edges in $\mathcal{D}(P) = \{P_1, P_2, \ldots, P_n\}$. Each edge P_j starts at vertex $S_{L(P_j)}$ and ends at S_x where x is the minimum value of mark status of all output vertices of P_j in DDG(P) or $(m + 1)$ if P_j has no output vertex.

The idea is to find the earliest step in which the problem can be started (S1, S2). This step is defined as the next step of the latest step of all problems that this problem depends on. S2 will always terminate because DDG(P) is a graph without cycle. S3 constructs the sequential diagram that satisfies the definition since from the way we mark the vertices in DDG(P), the index of start node is always greater than the index of the end node; there always exists an edge that connects S_i with S_{i+1} (if $(i < m)$).

4.4.2. Time constraints of sub-problems

Until now, we only know the time constraint T of the root problem P. The question is, if we can not solve P sequentially within the time T, we will need to solve the decomposition set $\mathcal{D}(P)$. Hence we need to calculate the time constraint for each $P_i \in \mathcal{D}(P)$.

We define *the time constraint of a step* S_i as the amount of time in which all problems $P_k \doteq < S_i, S_{i+1} > \in E$ have been solved.

We define *the constraint guard coefficient* as the chance that the execution of the decomposed sub-problems could satisfy the overall user-specified time constraint T. Let α be *the constraint guard coefficient* $(0 < \alpha \leq 1)$, if the original problem is replaced by the decomposed sub-problems, all sub-problems are expected to be solved within αT time unit.

Let assume $T_1^s, T_2^s, \ldots, T_m^s$ are the time constraints of steps $S_1, S_2, \ldots, S_m$. In order to satisfy the overall time constraint T, the following conditions must hold:

$$\forall i, T_i^s > 0 \tag{4.2}$$

$$\sum_{j=1}^{m} T_j^s \leq \alpha T \tag{4.3}$$

In the worst case, (4.3) becomes:

$$\sum_{j=1}^{m} T_j^s = \alpha T \tag{4.4}$$

The time constraint T_i of problem P_i is determined based on the time constraints of steps:

$$T_i = \sum_{j=k}^{l-1} T_j^s \quad \text{where } P_i \text{ is the label of edge } < S_k, S_l > \tag{4.5}$$

There are many solutions satisfying (4.2) and (4.4). In section 4.3.1, we assume that the number and the characteristics of resources in the environment are unknown. We need to find the time constraint of each step which *maximizes the chance to find resources for sub-problems.* In the Grid, it is likely that when the level of requirement on the resources decreases, the probability to find a resource that satisfies the requirement increases. Therefore, we choose the following criterion: *find the time constraints of steps such that they minimize the maximum computation power required by all sub-problems P_i of a decomposition.*

Let $C \subset \mathbb{R}^+$ and $T \subset \mathbb{R}^+$ be the complexity space and the time constraint space of the problem. Let R_e be the resource characteristic space. If we only consider the computing power of resources, the resource characteristic space can be defined as $R_e \equiv \mathbb{R}^+$. For each problem P_i, we define the *resource function* g_i as $C \times T \mapsto R_e$ which maps a sequential solution of problem with the complexity c and the time constraint t to the requirement of resource $g_i(c,t)$. In the simple case where the complexity of problem is the total number of operations, $R_e \subset \mathbb{R}^+$ represents the number of operations/sec, the resource function can be evaluated as:

$$g_i(c,t) = \frac{c}{t} \tag{4.6}$$

We need to find $T_1^s, T_2^s, \ldots, T_m^s$ satisfying the conditions (4.2) and (4.4) such that:

$$[T_1^s, T_2^s, \ldots, T_m^s] \quad = \quad \arg\min\max\{g_1(\mathcal{C}(P_1), T_1), g_2(\mathcal{C}(P_2), T_2), \ldots, g_n(\mathcal{C}(P_n), T_n)\} \tag{4.7}$$

where $\mathcal{C}(P_i)$ is the complexity of problem P_i, T_i is the time constraint of problem P_i that satisfies (4.5).

This is a min-max problem with constraints. Now we consider a special case where each problem spans exactly one step (i.e. n=m). The solution to (4.7) can be obtained by considering the complexity $\mathcal{C}(P_i)$ as the "weight" for the time constraint T_i^s:

$$T_i^s = \frac{\mathcal{C}(P_i)}{\sum \mathcal{C}(P_j)} \alpha T \tag{4.8}$$

Although solving min-max problems as in Eq.(4.7) is not the objective of this dissertation, we have developed a generic algorithm to find an approximate solution for Eq.(4.7). Details on this algorithm are described in appendix A.

4.5. Summary

Solving problems within a required time bound is a hard question. It is event more difficult to find a feasible solution on the Grid computing environments where resources in the pool can join or leave in time.

We have presented a parallelization scheme as a feasible approach to the time constrained problems on the Grid. The scheme consists of a decomposition tree defining possible decompositions of a problem into sub-problems and the decomposition dependency graph showing the relative order of execution of sub-problems. The scheme provides a way for programmers to specify their time constrained applications.

An algorithm based on the decomposition tree was constructed, showing how time constrained problems can be solved. It can be designed as a supporting framework or can be integrated into programming languages. The parallel object model can be used to build a framework for the parallelization scheme. We will describe this framework in chapter 8.

Part II

The ParoC++ Programming System

Chapter 5

Parallel object C++

5.1. ParoC++ programming language

ParoC++ is an extension of C++ that implements the parallel object model as defined in chapter 3. We try to keep this extension as close as possible to C++ so that programmers can easily learn ParoC++ and that existing C++ libraries can be parallelized using ParoC++ without too much effort.

We claim that *all C++ classes with the following restrictions can be implemented as parallel object classes without any changes in object's semantic*:

- All data attributes of object are protected or private.
- The object does not access any global variable.
- There is no user-defined operator.
- There is no method that returns the memory address references.

These restrictions are not a major issue in object-oriented programming and in some cases they can improve the legibility and the clearness of programs. The restrictions can be worked around by adding get/set methods to access data attributes and by encapsulating global data and shared memory address variables in other parallel objects.

Since parallel object is a generalization of the sequential object, so unless the term "sequential object" is explicitly specified, we refer our *parallel object* as *object*.

5.1.1. ParoC++ parallel class

Developing ParoC++ programs mainly consist of designing and implementing parallel classes. The declaration of a parallel class begins with the keyword **parclass** following the class name and the optional list of derived parallel classes separated by a comma (","):

```
parclass myclass {...};
```

or

```
parclass myclass:  baseClass1, baseClass2,...  {...};
```

As sequential classes, parallel classes contain methods and attributes. Method accesses can be *public*, *protected* or *private* while attribute accesses must be either *protected* or *private*. For each method, the user should define the invocation semantics. These semantics, described in section 3.4, are specified by two keywords, one for each side:

- Interface side:

  `sync`: Synchronous invocation. This corresponds to the traditional way to invoke methods and is the default value. For example:

  `sync void method1();`

  `async`: Asynchronous invocation. For example:

  `async int method2();`

- Object side:

  `seq`: Sequential invocation. This is the default value. For example:

  `seq void method1();`

  `mutex`: Mutex invocation:

  `mutex int method2();`

  `conc`: Concurrent invocation. The invocation occurs in a new thread.

The combination of the interface and the object-side semantics defines the overall semantics of a method. For instance, the following declaration defines an asynchronous concurrent method that returns an integer number:

```
async conc int mymethod();
```

It has to be mentioned that as in C++ language, multiple inheritance and polymorphism are supported in ParoC++. A parallel class can be a stand-alone class or it can be derived from other parallel classes. Some methods of a parallel class can be declared as overridable (**virtual** methods).

5.1.2. Object description

The object description used to describe the resource requirements is declared along with parallel object constructor statement. Each constructor of a parallel object associates with an OD that resides directly after the argument declaration between "@{...}". An OD contains a set of expressions on the reserved keywords **power** (for the computing power), **network** (for the communication bandwidth between the object server and the interface), **memory** (for the memory) and **host** (user-specified resource). Each expression is separated by a semi-colon (";") and has the following format:

`[power | memory | network]` [>=|=] <*number expression 1*>
["**?:**" *number expression 2*];
or `host` = <*string expression*>;

The *number expression 2* phrase is only used in non-strict OD items to describe the low-bound of the acceptable resource requirements. The existence of host expression will make all other expressions be ignored.

Example: the constructor for the parallel object *MyObj*:

```
parclass MyObj
{
public:
      MyObj(float P) @{ power=P; memory=60; };
      ...
};
```

Following this constructor, the *MyObj* object will be created on a resource with at least *P* MFLOPS (parameterized OD) and a memory available of at least 60MB.

The OD is used by the ParoC++ run-time system (chapter 7) to find a suitable resource for the parallel object. Matching between OD and resources is carried out by the multi-layer filtering technique: first, each expression (item) in OD will be evaluated and categorized (e.g., power, network, memory). Then, the matching process consists of several layers; each layer filters single category within OD and performs matching on that category. Finally, if the OD can pass all filters, the object is assigned to that resource.

5.2. Parallel object manipulation

5.2.1. Parallel object creation and destruction

ParoC++ manages the parallel object life time by an internal object counter. This counter defines the current number of references to the object. A counter value of 0 will make the object be physically destroyed.

Syntactically, the creation and the destruction of a parallel object are identical to those of C++. A parallel object can be implicitly created by just declaring a variable of the type of parallel object on stack or using the standard C++ **new** operator. When the execution goes out of the current stack or the **delete** operator is called, the reference counter of the corresponding object will be decreased.

The object creation process consists of several steps: locating a resource satisfying the OD (resource discovery), transmitting and executing the object code, establishing the communication, transmitting the constructor arguments and finally invoking the corresponding object constructor. Failures on the object creation will raise an exception to the caller. Section 5.2.5 will describe the ParoC++ exception mechanism.

5.2.2. Inter-object communication: method invocation

The conventional way to communicate between distributed components in ParoC++ is through method invocations. The invocation semantics are specified during the class declaration (section 5.1.1). The user can add optional argument descriptors to each argument to provide additional information for ParoC++ to marshal data: the argument is input or output or both; the number of elements inside the argument (in the case of array); the custom procedure to marshal/demarshal data, etc. In the current prototype of ParoC++, all standard C++ data types are automatically marshaled. For user-defined data types, the user should also specify the function to marshal data by an optional descriptor [**proc**= <*marshal function*>]. If an argument of method is an array, it is also necessary that the user provide a hint on the number of elements by the expression [**size**= <*number expression*>].

For example, to define a method *Foo* that has two arguments: an array of integer *data* (input and output) and the size *n* of data:

```
parclass MyObj {
    ...
    void Foo([in, out, size=n] int *data, int n);
    ...
};

//main program....
...
    MyObj obj;
    int data[10];
    obj.Foo(data,10);
...
```

The current prototype of ParoC++ implements the communication using the TCP/IP socket and the Sun XDR as its data representation. All data transmitted over the network conforms to the XDR format.

5.2.3. Intra-object communication: shared data vs. event sub-system

In parallel objects, there are two ways for concurrent operations to communicate: using *shared data attributes* of the object or using *the event sub-system*. Communication between operations using shared attributes is straightforward because all operations on the same object share the same memory address space. The programmer should verify and synchronize the data access manually. Nevertheless, ParoC++ provides different method invocation semantics (see section 3.4) for the programmer to control the level of concurrency of data access inside each parallel object.

An alternative method is to communicate via the *event sub-system*. In ParoC++, each parallel object has its own event queue. Each event is a positive integer whose semantic is application dependent. A parallel object can raise or can wait for an event in its queue. Waiting for an event will check on the object event queue whether the event has arrived or not. If not, the execution of the current method will be blocked until the event occurs in the queue. An event "n" can be raised by the operation **eventraise(n)**. A method can wait for an event by the operation **eventwait(n)**. Raising an event in a parallel object does not affect the waiting-for-event in other parallel objects.

The following example demonstrates the use of the event sub-system to signal the arrival of the new data of the parallel object *MyObj*. The concurrent method *SetData* will store the data to the object and then raise an event. The method *WaitData* will be blocked until the *SetData* method has been completed.

```
 parclass MyObj
 {
 ...
     conc async void SetData(...);
     conc sync void WaitData();
...
 };

 void MyObj::SetData(...)
 {
    //store the data here....
    ...
    eventraise(1);
 }
 void MyObj::WaitData()
 {
    eventwait(1);
 }
```

The event sub-system is a very powerful feature to deal with signaling and synchronizing in distributed environments. For instance, it is used in conjunction with the shared data attributes to notify the changes in data states during the concurrent invocations of read/write operations. It can also be used to tell the others about occurrence of failure or the changes inside the executing environment.

5.2.4. Mutual exclusive execution

When concurrent invocations occur, some parts of executions might access an attribute concurrently. To deal with this situation, it is necessary to provide a mutual exclusive mechanism. ParoC++ supports this mechanism by providing the keyword **mutex**. Inside a given object, all block of codes starting with the keyword **mutex** will be executed mutual exclusively. In the example bellow, methods *Set* and *Get* on the attribute *val* are performed atomically.

```
 parclass MyObj
 {
 ...
     conc async void Set([in] int data[10]);
     conc async void Get([out] int data[10]);
... private:
    int val[10];
 };

 void MyObj::Set(int data[10])
 {
    ....
    //MUTEX1
    mutex {
        for (int i=0;i<10;i++) val[i]=data[i];
    }
    ...
  }
 void MyObj::Get(int data[10])
 {
    ....
    //MUTEX2
    mutex {
        for (int i=0;i<10;i++) data[i]=val[i];
    }
    ...
}
```

5.2.5. Exception support

The exception is an efficient mechanism for handling errors. Instead of handling each error separately based on the "error code" returned by a function call, exception mechanisms allow the user to filter and centrally manage the errors within the calling stacks. Upon an error occurs, the user will "throw" an exception which will be caught somewhere else.

The exception mechanism in sequential applications where all components run within the same memory address space is fairly simple since the compiler just need to pass a pointer to the exception from the place where it is thrown to the place where it is caught. However, in the heterogeneous distributed environment where each component is executed in a separate memory address space and the native data representations are different due to the hardware heterogeneity, the propagation of exception back to a remote component is more difficult. The supporting system should know the type of exception, marshal the exception data, transmit the data and demarshal the data on the remote component. On the remote component, the exception should be en route until it is caught by the user.

ParoC++ supports transparent exception propagation: the exception occurs in a parallel object will be automatically propagated back to the remote caller (Fig. 5.1). The current prototype ParoC++ allows the following types of exceptions: all standard types (except opaque type), parallel object types (user customizable exceptions) and the "*paroc_exception*" type (system exception).

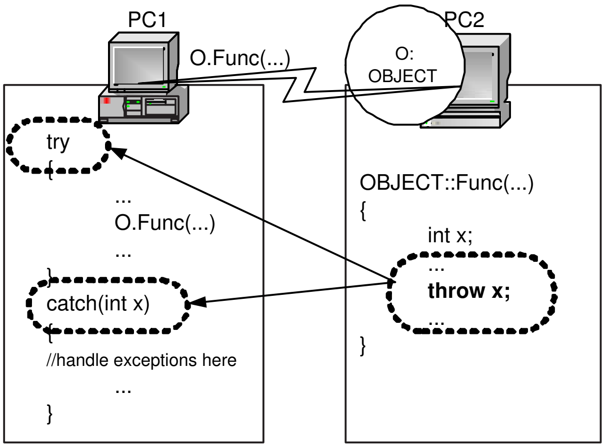


Figure 5.1: ParoC++ exception handling: PC1 makes a method call to object O on PC2. The exception occurred on PC2 will be handled on PC1 with the pair "try" and "catch" on PC1

Two invocation semantics affect the propagation of exception: with synchronous invocations, the exception will be immediately propagated back to the caller; with asynchronous invocations, since the caller does not wait for the results, the exception will be propagated back to the caller the next time the caller accesses that object.

Beside the user exceptions, ParoC++ uses a special exception type *paroc_exception* to notify the user about the system failure:

- Parallel object creation fails due to the unavailability of suitable resources, an internal error on ParoC++ services, or the failures on executing the corresponding object code, etc.

- Parallel object invocation fails due to the network failure, the remote resource down, etc.

All exceptions of type parallel object are propagated by reference, i.e., only the interface of the exception is sent back to the caller. Other exceptions are transmitted to the caller by value.

5.3. ParoC++ compiler

The compilation process is illustrated in Fig. 5.2. The ParoC++ compiler contains a ParoC++ parser which translates the ParoC++ code to the ANSI C++ code; the ParoC++ service libraries that provide APIs for accessing various run-time services such as communication, resource discovery and object allocation, etc.; and an ANSI C++ compiler to generate binary executables from the C++ code and the service libraries. The ParoC++ compiler generates a main executable and several object executables for each ParoC++ application. The main executable provides an entry point for the user to run the application. Object executables are not used directly by the user but they are accessed by the ParoC++ run-time system during the parallel object creation.

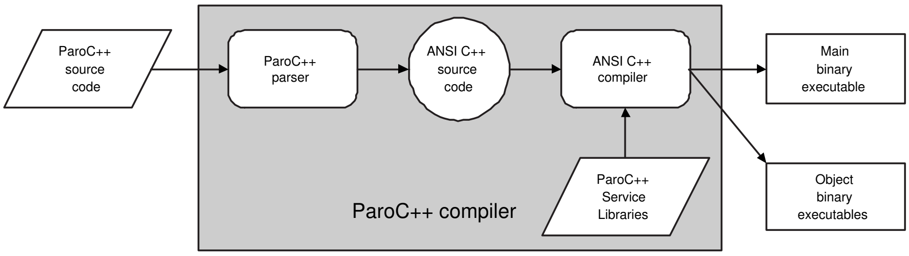


Figure 5.2: ParoC++ compilation process

5.4. Putting together

We show in this section an example of writing a ParoC++ program in the following scenario: the user sits in front of his computer, create two "remote" objects. He then, from the main program, passes the interface of one object to the other. The other object will perform some operations on the object provided by the user. The "object" in the example is represented for an integer and the operation is to increase the integer by a value remotely stored in the other integer.

5.4.1. Programming

Figure 5.3 shows the declaration of a parallel class in ParoC++. From the language aspect, this part contains the major differences between the ParoC++ and the C++. However, the example shows that ParoC++ syntax is very similar to the C++ class declaration except some new keywords (in bold letters). A parallel class consists of constructors (line 3-4), destructor (optional), interfacing methods (public) (line 5-7), and a data attribute (private) (line 9).

File: *integer.ph*

```
1:    parclass Integer {
2:    public:
3:        Integer (int wanted, int minp) @{power>=wanted ?: minp;};
4:        Integer(char *machine) @{host=machine;};
5:        seq async void Set(int val);
6:        conc int Get();
7:        mutex void Add(Integer &other);
8:    private:
9:        int data;
10:   };
```

Figure 5.3: ParoC++ example: parallel class declaration

The user defines a parallel class called `Integer` starting with the keyword ***parclass*** (line 1). Two constructors (line 3 and 4) of `Integer` are both associated with two ODs which reside right after the argument declaration, between "@{...}". The first OD (line 3) specifies the parameterized high level requirement of resource (i.e. computing power). The second OD (line 4) is the low-level description of the location of resource on which the object will be allocated.

The invocation semantics are defined in the class declaration by putting corresponding keywords (*sync*, *async*, *mutex*, *seq*, *conc*) in front of the method declaration. In the example, the `Set` method (line 5) is sequential asynchronous, the `Get` method (line 6) is concurrent and the `Add` method (line 7) is mutual exclusive execution (atomic execution). Although it is not shown in the example but the user can also use standard C++ features such as *virtual*, *const*, or inheritance with the parallel class.

The implementation of the parallel class `Integer` is shown in Fig. 5.4. This implementation does not contain any invocation semantic and looks similar to a C++ code except at line

File: *integer.cc*

```
1:    #include "integer.ph"
2:    Integer::Integer(int wanted, int minp)
3:    {
4:    }
5:    Integer::Integer(char*machine)
6:    {
7:    }
8:    void Integer::Set(int val) {data=val;}
9:    {
10:      data=val;
11:   }
12:   int Integer::Get()
13:   {
14:      return data;
15:   }
16:   void Integer::Add(Integer &other)
17:   {
18:      data=other.Get();
19:   }
20:   @pack{Integer};
```

Figure 5.4: ParoC++ example: parallel object implementation

20 where we provide a directive "pack" to tell the ParoC++ compiler the place to generate the parallel object executable for `Integer` (*integer.cc*).

The main ParoC++ program (Fig. 5.5) looks exactly like a C++ program. Two parallel objects of type `Integer` `o1` and `o2` are created (line 3). The object `o1` requires a resource with the desired performance of 100MFlops although the minimum acceptable performance is 80MFlops. The object `o2` will explicitly specify the resource location (local host). After the object creations, the invocations to methods `Set` and `Add` are performed (line 4-5). The invocation of `Add` method shows an interesting property of the parallel object: the object `o2` can be passed from the main program to the remote method `Add` of parallel object `o1`. Lines 8-11 illustrate how to handle exceptions in ParoC++ using the keyword pair `try` and `catch`. Although o1 and o2 are distributed objects but the way to handle the remote exceptions is the same as in C++.

5.4.2. Compiling

We generate *two executables*: the main program (main) and the object code (integer.obj). ParoC++ provides the command "parocc" to compile ParoC++ source code.

Compile main program We use the following command:

```
parocc -o main integer.ph integer.cc main.cc
```

File: *main.cc*

```
#include "integer.ph"
int main(int argc, char **argv) {
    try { Integer o1(100,80), o2("localhost");
        o1.Set(1); o2.Set(2);
        o1.Add(o2);
        cout<<"Value="<<o1.Get();
    }
    catch (paroc_exception *e) {
        cout<<"Object creation failure";
        return -1;
        }
    return 0;
}
```

Figure 5.5: ParoC++ example: the main program

Compile the object code Use parocc with option ”-object” to generate the object code:

```
parocc -object -o integer.obj integer.ph integer.cc
```

The user has to compile the declaration of parallel class (integer.ph) explicitly. The user can also generate intermediate code (.o) that can be linked using a C++ compiler by using the option ”-c” (compile only) with ”parocc”.

5.4.3. Running

After the two executables are generated, we need to create the object map file named *object.map* that will be used by the code manager service (chapter 7). The object map file contains all mappings between (object name, platform) and the executable location. The executable location can be an absolute path or an URL (HTTP or FTP). We assume to compile the program on Linux machines and to put the executable on the web server at the following address:

http://icwww.epfl.ch/∼tanguyen/paroc/

The object map file should look like:

```
Integer Linux http://icwww.epfl.ch/∼tanguyen/paroc/integer.obj
```

If you compile the object code for another platform (e.g. Solaris), just add a similar line to *object.map.*

Now it is ready to run the program. ParoC++ provides the command ”parocrun” to do that. From the local machine, the user starts the ParoC++ main program by executing:

```
parocrun object.map main
```

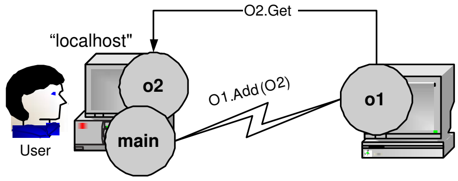


Figure 5.6: Three objects "O1", "O2" and "main" are executed in separated memory address spaces. The execution of "o1.Add(o2)" as requested by "main"

Figure 5.6 shows the execution of *Integer::Add* method on line 4 in Fig. 5.5 of the example. The system consists of three running processes: the main, object `o1` and object `o2`. The main is started by the user. Objects `o1` and `o2` are created by the main. Object `o2` and the main program run on the same machine although in two separate memory address spaces; object `o1` runs on a remote machine. The main invokes the "`Add`" method on `o1` with the interface `o2` as an argument. Object `o1` will then connect to `o2` automatically and invoke the method "`Get`" on `o2` to get the value and to add this value to its local attribute "data". ParoC++ system manages all object interactions in a transparent manner to the user.

5.5. Summary

We presented in this chapter the language aspects of Parallel Object C++ (ParoC++). ParoC++ is an extension of C++ language that implements the parallel object model (chapter 3). Rather than presenting all aspects of the ParoC++ language, we only focus on the extension part of ParoC++ for parallel objects. We describe features of parallel objects in ParoC++: declaration, implementation, manipulation, concurrent synchronization, exception handling, etc. The overall architecture of ParoC++ compiler is also presented. A complete example of ParoC++ illustrates how to write a parallel object program in ParoC++. ParoC++ allows applications to put the object executables on the web so that remote machines, through the ParoC++ runtime system, can download and execute the parallel object automatically and transparently to the user. ParoC++ programming is simple and it can be used as a good portal to achieve the seamless power of the Grid.

The language aspect of ParoC++ is just one link of the chain user-application-infrastructure. The other link that completes the chain is the supporting infrastructure-the ParoC++ runtime system that will be described in chapter 7.

Chapter 6

Data intensive computing in ParoC++

6.1. Introduction

Data-intensive high performance computing applications can be grouped into two categories. The first category refers to applications that need to manage and to access a huge data set. For instance, Large Hadron Collider (LHC) [17] can produce annually petabytes of data. Nevertheless, for a group of scientists, only a part of this data really needs to be efficiently accessed, modified and computed while the rest are kept untouched. Users are usually not located at the site where the data are stored but they rather work in some collaborative environments over the world. Many techniques for managing data movement have been developed. In [1, 18, 70, 73], the authors describe some issues in the context of Data Grids such as the GridFTP [2] mechanism for data transport and replica management.

The second category of data-intensive computing is communication-intensive applications. These applications require that efficient communication be achieved in addition to computational tasks. We are more facing with the communication intensive issue rather than the large data management issue. Some techniques such as data caching [54, 78], replica catalog [73] have been implemented to deal with communication intensive problems.

In this chapter, we focus on the second class of data intensive computing, i.e., parallel computing with communication intensive. In the framework of ParoC++, we propose a technique called *passive data access* for efficient data movement in high performance data intensive computing.

6.2. Data access with ParoC++

6.2.1. Passive data access

Communication intensive applications usually use two main techniques to improve the performance of data movement: moving the computation toward the data source or moving the data toward the computation process.

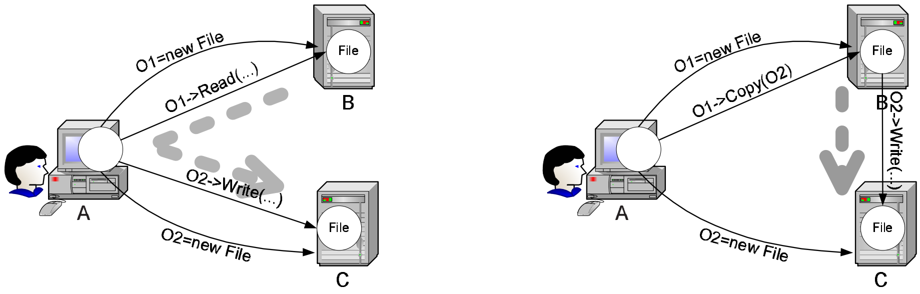
In ParoC++, moving the computation to the data source can be implemented by allocating parallel objects ”near” the data source in order to fulfill the object description requirements. However, if the computation involves multiple distributed data sources, it is often difficult, even impossible, to locate a resource near all data sources. Another situation is when a heavy computation task is required and no resource near the data source can provide the satisfying computing power. In such situations, moving the computation toward the data source is not appropriate and we have to move the data toward the computation process.

A widely used technique to increase the performance of data movement in parallel computing is to overlap computation and communication. A number of studies has been focused on this issue such as asynchronous sending and combining non-blocking receive with polling in MPI [72] and PVM [34], pipelining large data in C++// [33], etc.

We propose a technique called **passive data access** (or passive access for short) to improve the overlap between computation and communication. Passive access allows a data source to directly store the data to a user-specified address without intervening in other user's operations. It is the data source that decides when to initialize data transferring. This is similar to DSM systems [8, 67]. However, while DSM aims at providing a model for parallel programming, passive access is a technique to efficiently access data in the Grid or distributed environments. In addition, the user-specified address in passive access is not only limited to the memory, but it can also be a secondary storage, a network device or other media.

Passive access in ParoC++ is described as follow: a computational object C requires a series of data pieces from one or several distributed data source objects $S_1, S_2, \ldots S_n$. The data that C needs is stored in a parallel object D. Object D and object C can be the same or they can be different. Object C will invoke asynchronously the method ”request-for-data” on the source objects with an argument containing the reference of the data destination object D. Then it continues its operation. At the source S_i, the ”request-for-data” method will prepare the data and initialize the data storing to object D (attributes) directly without notifying C (if C and D are the two different objects). When the data is needed on C, C will access directly D to acquire the needed data.

Figure 6.1 shows an example of remote copying file from machine B to machine C. The user on machine A will create two objects of type ”File” on B and C. Then he performs copying. In the example 6.1(a), the user uses ”Read” to read from B and ”Write” to write data to the destination file on C. Example 6.1(b) shows passive data access in which the user implements the method ”Copy” with a destination file object as the argument. The ”Copy”



(a) Traditional approach

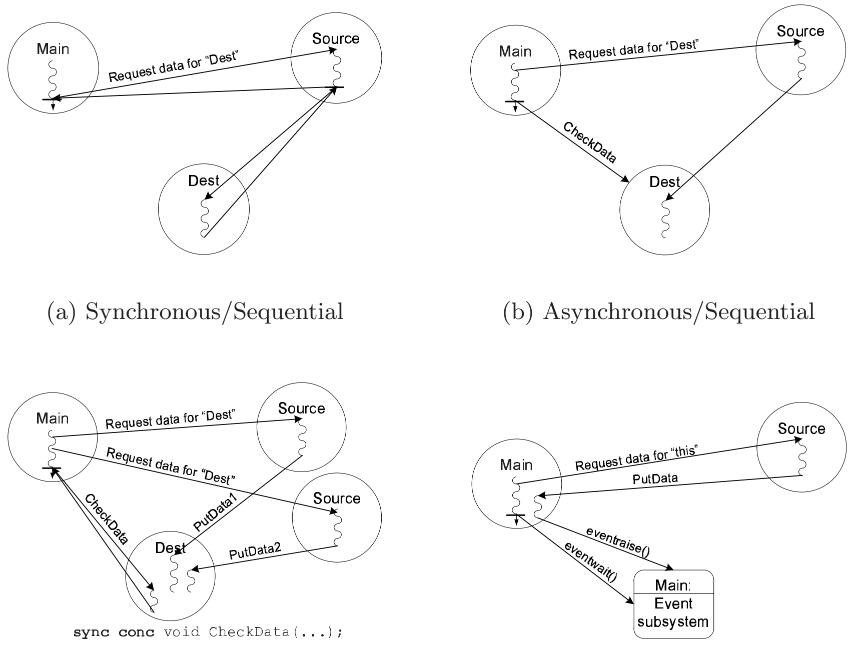
(b) New approach

Figure 6.1: Passive data access illustration

operation on B with a reference to C as the argument will make the data transfer directly from B to C without going through the user machine.

Figure 6.2 shows different possibilities to access data using passive access in ParoC++. ParoC++ provides various mechanisms for data access: synchronous, asynchronous, sequential, mutex and concurrent. In the first 2 cases (6.2(a) and 6.2(b)), the data is acquired from a single data source. Figure 6.2(a) shows a traditional method to synchronously access data: the "main" requesting data for the parallel object "dest" is blocked until the data arrive at "dest" from the source. Synchronous access is usually not a preferable method in parallel computing because it reduces the possibility to overlap between computation and communication. In figure 6.2(b), the "main" does not wait for the data. Instead, it calls an asynchronous invocation on the data source and continues the execution. At a certain time, when the "main" needs the data, it checks for the data at the "dest" and waits for the availability of this data. Figure 6.2(c) introduces the concurrent data access. Data can be obtained from several data sources concurrently. When the "main" needs the data, it just invokes a "check" method on the parallel object "dest". The check runs concurrently with the data storing process. Figure 6.2(d) illustrates the situation where "dest" and "main" are the same object. This illustrates the ability to store data directly into the computational memory address space.

One important issue of passive access is how the data consumer knows if the data is ready for use or not. The arrival of data can be triggered by raising an event (event sub-system). In Figure 6.2(d), when the computational task requires the data, it just waits for an "arrival" event which will be produced by the set-data methods ("PutData" method in the example). The event sub-system is only one way to check for the availability of data. An alternative way is to use a "flag" attribute as the meta-information (status) of the data. "PutData" will change the flag while the computational method will check on this flag the status of data arrival.



```
sync conc void CheckData(...);
```

(a) Synchronous/Sequential

(b) Asynchronous/Sequential

(c) Asynchronous/Concurrent without mutex on multiple data sources

(d) Passive access: check for data arrival

Figure 6.2: Passive data access in ParoC++

6.2.2. Data Prediction

Passive access allows users to request for the data prior to the computation. In other words, users can predict the data needed in their applications and then signal the data source to prepare for their data in advance. The computation is not blocked during this preparation. Therefore, passive data access can reduce the time waiting for data and gives a better overlap between computation and communication.

In many applications, the data is not raw data but it is rather computed. Preparing data can be a time-consuming process. So passive access offers an efficient method to increase the overlap between the data preparation and the computation.

6.2.3. Partial data processing

When users need to process a large amount of data and the computation can take place when a portion of the data is available, one possible way to increase the overlap between computation and communication is to split the data into smaller chunks and transfers these chunks one by one. This idea can be implemented in ParoC++ using passive data access with multiple data buffer at the computational objects. Each buffer corresponds to a data chunk. When a given amount of the data in the current buffer has been used up, requests for data will be invoked on the data source to prepare for the next data. Then the computation continues on the remaining buffer. During this computation, the new chunks are expected to arrive, avoiding the computational process to be blocked when all the data in the buffer has been consumed.

6.2.4. Data from multiple sources

The data needed for the computation may come from multiple data sources or data providers. Data management and aggregation is a complex process that might increase the cost of parallelism. The object-oriented framework in ParoC++ allows the needed data to be acquired and be aggregated automatically from multiple data sources. Each data source is responsible for putting its data to a proper place by invoking an appropriate method on the destination object.

In CAVE [23] virtual reality system, for instance, the user needs the data from several cameras, sensors in order to produce a virtual interactive environment. One possible way using ParoC++ is to create computational parallel objects. Each object will introduce itself to the data sources. At a data source, as the data become available, the attribute of the computational object will be updated by the corresponding method invocation ("set" method). Doing this way makes the data transfer transparent to the data consumer.

6.3. Summary

The chapter presented an approach for efficient data access in distributed data intensive applications called passive data access. The main difference between passive data access and other methods is the ability for an entity (usually a data source) to initiate the transfer and to store data directly to the user-specified address without interrupting other user's operations running in that address.

Passive data access with ParoC++ provides a new way to develop distributed communication intensive applications. The user focuses on constructing different kinds of parallel objects and on defining the roles, the resource requirements, the relationships among parallel objects rather than on defining the messages exchanged between processes. Data can be actively brought and synthesized from data sources to data consumer objects in advance and in an efficient way. Passive data access also improves the overlap between communication and computation which leads to the improvement of the overall performance of parallel applications.

Some experiments with ParoC++ data access will be described in chapter 10.

Chapter 7

ParoC++ runtime architecture

7.1. Overview

The ParoC++ runtime system is used to execute ParoC++ applications. We do not intend to address all issues of Grid computing, instead, we will provide essential services for ParoC++ applications and a mechanism to integrate other low-level Grid computing toolkits such as Globus [28, 29] or XtremWeb [25] into ParoC++. Each ParoC++ service is implemented as a parallel object. We define a high-level abstraction of the service functionalities though the object interface without any specification on how to implement it. The default implementation is provided as a reference. The user can implement his own service such as the Globus-based service by replacing the object implementation or by inheriting the new service (new object) from the existing service (existing object). The latter possibility is recommended since it corresponds better to the object-oriented programming style. A new system can be integrated automatically into the ParoC++ environment in a Plug-and-Play flavor that existing ParoC++ applications can immediately use without being recompiled.

Figure 7.1 illustrates the role of ParoC++ in the Grid environments. Rather than providing low-level services for direct access to computational resources, the ParoC++ runtime system consists of an abstraction of services and the customizable and extensible implementations of the abstraction on the Grid-enabled toolkits. Hence we can use ParoC++ to glue Grid tools for executing HPC applications.

7.2. ParoC++ execution model

The execution model of ParoC++ applications is MIMD (Multiple Instructions Multiple Data stream). Thank to the asynchronous and concurrent invocations, the executions on different parallel objects can simultaneously occur. The executions of concurrent methods on the same object are also performed in parallel.

Unlike the SPMD (Single Program Multiple Data stream) model where instructions are

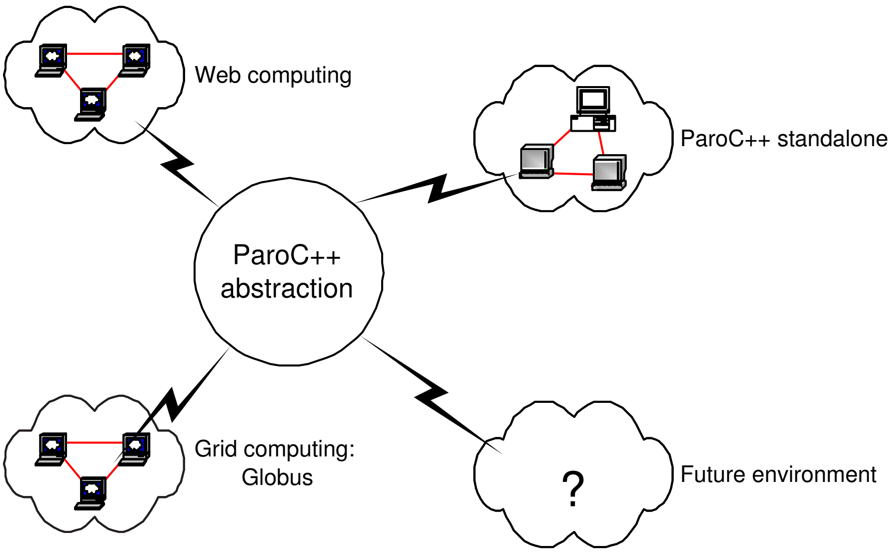


Figure 7.1: ParoC++ as the glue of low level Grid toolkits

stored inside a single program, ParoC++ applications are truly MIMD. Each application consists of a main program and several object executable files. Object files store the implementation of parallel objects which are managed by the ParoC++ runtime system. In ParoC++, an object implementation can reside in several object files compiled for different hardware architectures. The availability of an object implementation for a specific architecture will be considered during the object allocation phase. The difficulty of using multiple code files is that we need to either explicitly or implicitly manage the files. The ParoC++ system manages this automatically and transparently to the user. In the Grid, splitting the whole application into multiple code files has two advantages. First, we only need to transfer the code over the network that is really used to perform the task. Second, each part of the application can be optimized for several specific architectures and hence it can increase the robustness of applications.

When the user starts the ParoC++ main program, there is only one execution stream. Then, parallel objects are created and destroyed dynamically as if we do on sequential objects. The parallel object creation accesses ParoC++ services to perform resource discovery, to download and to launch suitable object file on the remote resource, to establish the interface-object connection, etc. The object creation process can be invoked anywhere inside the application: in the main program or in a remote parallel object, transparently to the user. The object destruction occurs when the execution goes out of the stack where the parallel object is declared or the ”delete” operator on the object is explicitly called (in the case the object is created by the ”new” operator).

Since the parallel object is shareable, each parallel object is only physically destroyed when there is no more reference to the object. This feature is automatically managed by the

ParoC++ system.

7.3. Essential ParoC++ services

We do not intend to build a complete system to deal with all issues of the Grid environment. Instead, we will construct high level services that are essential to run ParoC++ applications. These high level services should provide:

- A high-level abstract and uniform interface for ParoC++ applications without any specific implementation details.
- A customizable and extensible implementation that lays on other Grid computing toolkits.

Figure 7.2 shows the architecture of ParoC++ services. The *ParoC++ application* is on the top of the architecture. It uses the standard interface defined by the *ParoC++ essential service abstraction* to perform operations on parallel objects such as object creations, object destructions, method invocations, etc. This abstraction layer defines the desired functionalities of services, not how these services are implemented. The *customizable service implementation* layer provides different implementations of the abstraction on different environments.

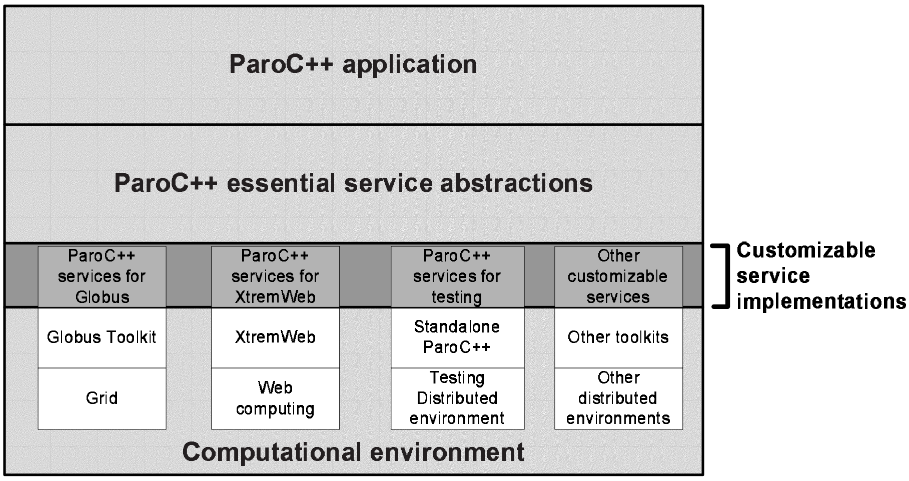

ParoC++ application
ParoC++ essential service abstractions
ParoC++ services for Globus
ParoC++ services for XtremWeb
ParoC++ services for testing
Other customizable services
Customizable service implementations
Globus Toolkit
XtremWeb
Standalone ParoC++
Other toolkits
Grid
Web computing
Testing Distributed environment
Other distributed environments
Computational environment

Figure 7.2: ParoC++ layer architecture

Our favor in building the ParoC++ runtime system is to glue other Grid or distributed environments such as Globus, XtremWeb, etc. into a single virtual system that is exploitable by ParoC++ applications. When a new environment is integrated into ParoC++, its resources will be automatically available to all ParoC++ applications.

From the execution model in section 7.2, ParoC++ services focus on supporting the dynamic creation of parallel objects. Parallel objects can reside in different executable files.

The same parallel object can be compiled for several hardware architectures. Therefore, we need a service to manage different parallel object code files. We call this service the *ParoC++ code manager service* (section 7.4). Rather than specifying the exact location of resource, each parallel object describes the characteristics of the resource where the object will live. ParoC++ provides the *ParoC++ resource discovery service* to find a suitable resource for the object based on the object description (section 7.6). Besides the object description, the resource discovery service also accesses the code manager service to acquire information about supported platforms for the given object. When a suitable resource is identified, the system should be able to launch the object server on the remote resource and to manage the execution status. This functionality is provided by the *ParoC++ object manager service* (section 7.7).

Sometimes the user needs to log some information. Writing out messages in a readable way to the user's screen is a problem in distributed environments. The ParoC++ runtime system provides the *ParoC++ remote console service* that sends back all log information to the local console where the user starts the application. Section 7.5 describes this service.

ParoC++ services are accessible via the service *access points* through the TCP/IP protocol. Currently, each access point consists of the name of the machine where the service is running and the port on which the service listens for requests.

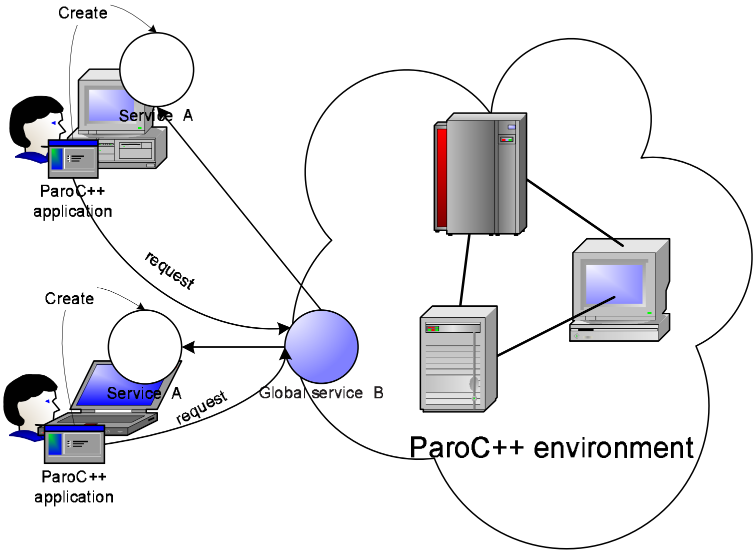


Figure 7.3: Global services and application scope services in ParoC++. Users create application scope services. Global services access application scope services to perform application specific tasks.

We categorize ParoC++ services into two classes: global services and application scope services. Figure 7.3 gives an illustration of two kinds of services.

Global services: The global services provide common functionalities for all ParoC++ applications. These services must run on each resource (a computer, a cluster, a supercomputer, etc.). They exist permanently even if there is no ParoC++ application running. When a parallel object is remotely created, the ParoC++ runtime system will automatically initialize the global service access point on the object. That makes the global service accessible from any parallel object inside the application. Global service access points do not need to be the same on all parallel objects within the application. The ParoC++ system can provide different global access points for different objects. The object manager and the resource discovery are two ParoC++ global services.

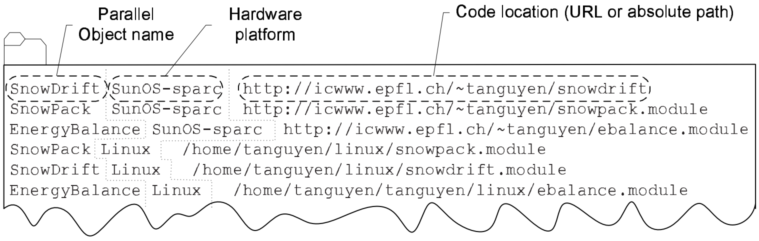
Application scope services: Differ from other systems, we introduce a new type of services called the *application scope services*: *services that serve specifically for each application.* Application scope services are instantiated by the user who runs the application. These services can be used by any application components or by other services to perform application-related tasks. Instances of these services are associated with each application. They are started when the application starts and are terminated when the application terminated. All components of the same application have the same access point to the application scope services. The ParoC++ code manager and the ParoC++ remote console are application scope services.

One important feature of the ParoC++ service architecture is the inter-operability between global services and application scope services. Global services can dynamically access to application scope services to obtain necessary application specific information for performing the requested tasks for the application. ParoC++ manages these inter-operabilities by defining the well-known interfaces for accessing services. Global services only know the application scope service abstractions. They do not know how the application scope services are implemented. The implementation of services can be customizable and extensible by users.

7.4. ParoC++ code manager service

The code manager service is an application scope service used by other global services such as the resource discovery (section 7.6) or the object manager (section 7.7) to locate the object executable for a specific object on a specific platform. This service is accessible via the parallel object interface *CodeMgr*.

Each ParoC++ application has its own object mappings, so the code manager service is defined as an application scope service. This service is instantiated separately for each application. By default, *CodeMgr* is started on the local machine where the user starts his application. The ParoC++ runtime system then makes it automatically available to all distributed components within the application. *CodeMgr* is not used directly by the programmer. It is internally used inside the system libraries to discover a suitable resource and to start the object execution on that resource.


Parallel
Object name
Hardware
platform
Code location (URL or absolute path)

```
SnowDrift SunOS-sparc  http://icwww.epfl.ch/~tanguyen/snowdrift
SnowPack  SunOS-sparc  http://icwww.epfl.ch/~tanguyen/snowpack.module
EnergyBalance SunOS-sparc  http://icwww.epfl.ch/~tanguyen/ebalance.module
SnowPack Linux   /home/tanguyen/linux/snowpack.module
SnowDrift Linux   /home/tanguyen/linux/snowdrift.module
EnergyBalance Linux   /home/tanguyen/tanguyen/linux/ebalance.module
```

Figure 7.4: Example of an object configuration file

When a ParoC++ application is started, the user needs to provide an object configuration file (Fig. 7.4) which lists all *object mappings* between (object name, supported platform) and the location of the object code. The object configuration file is a text file in which each line contains three fields separated by one or more spaces: the parallel object name, the platform and the location of the parallel object code compiled for this platform. The location can be an absolute path inside the local file system (in the case where the ParoC++ environment shares the same file system) or an URL that is accessible via some standard web protocols such as HTTP or FTP. Information in this object configuration file will be automatically registered to the *CodeMgr* via the method *RegisterCode* by ParoC++ when the user starts the application:

```
void CodeMgr::RegisterCode(char *objname, char *platform, char *codefile);
```

in which the user provides the parallel object name in *objname*, the platform on which the object code file was compiled (e.g. SunOS-sparc or Linux) in *platform* and the location of the code file in *codefile*. This information is the same as those in the object configuration file illustrated in Fig. 7.4.

The object mapping can be queried via the operation *QueryCode* where the user should provide the object name (*objname*) and the hardware architecture (*platform*) as the inputs; if the object code exists, its location is stored in the *codefile* argument and the method returns 1 (TRUE). If the object code is not available, the return value is 0 (FALSE) and the content of *codefile* is undefined.

```
int CodeMgr::QueryCode(const char *objname, const char *platform, char *codefile);
```

To get the list of supported hardware architectures for a specific object, the user uses the operation *GetPlatform* with two arguments: *objname* (the object name) as the input and *platforms* (list of platforms for *objname*, separated by a space) as the output. *GetPlatform* returns the number of supported platforms for the object.

```
int CodeMgr::GetPlatform(char *objname, char *platforms);
```

7.5. ParoC++ remote console service

Sequential C/C++ applications usually print out messages to the console using standard functions (or objects) such as *printf*, *puts* or the C++ iostream interface *cout*. However, on the Grid, to collect output messages from remote components, rather often the standard output and standard error files are redirected into a socket which at other end will be connected to the local standard output and standard error files. This approach does not work well in ParoC++ application because the place where the parallel object is created is usually not the local machine where the user starts the application. Therefore, the "local" console is not known by the parallel object.

We provide the remote console service that is accessible through the parallel object interface *RemoteLog* by all parallel objects to print back the message on the local user's console. *RemoteLog* is an application scope service that is started on the same machine where the user starts the application. The main operation of *RemoteLog* is to print out a message (string of characters) to the local console:

```
async virtual void RemoteLog::Log(char info[256]);
```

Log is a virtual method. The user can overwrite this method to implement his desired behavior (e.g. output the information to a file).

To facilitate the use of this service, ParoC++ provides a *printf*-like function that uses RemoteLog to print back the message to the user console:

```
void rprintf(const char *format,...);
```

7.6. Resource discovery

Resource discovery and reservation is an important service in the ParoC++ runtime environment. It is used to locate a suitable resource which satisfies the object description (OD) (see section 3.6 for OD) during the parallel object creation and to reserve the resource for that object. Unless the low-level OD, which explicitly specifies the resource (host) name, is used this service is accessed every time a parallel object is created.

7.6.1. Overview

Resource discovery is a hard but important problem for wide area resource management, typically of Grid environments. It is not efficient, even impossible for a user to manually find best resources for his application among thousands or even millions of nodes. The resource discovery can be stated as following: *within the computational environment, identify (some or all) available resources that satisfy user-specified constraints.*

The state-of-the-art of resource discovery systems can be classified into three approaches:

Broadcast/multicast approach: this is the traditional model used mainly in small flat networks such as LAN environments based on broadcast or multi-cast protocols. Each resource listens for a request message from the network. When the user looks for resources he just broadcasts or multi-casts the request to all (or a group of, in the case of multi-cast) resources. A willing-to-serve resource receives and verifies the request from the user with its local policy. If the request is matched, the resource replies back to the user with its information (location). This method is called active resource discovery. The advantage is the ability to automatically and dynamically discover resources inside the environment with the up-to-date dynamic information. However, the limitation of this approach is on the scalability of the system only to a small network (institution scale). Increasing the number of resources can seriously degrade the performance of the system. In wide area environments where resources can belong to different administrative domains, such as on the Internet, this model is not applicable.

Centralized approach: all information of resources is stored in one or several servers. The resource discovery process will query the server to find a suitable resource. The matching between user requirement and a resource can be performed on the central server or at the user site. In the latter case, the server only plays the role of an information provider. Centralized management is simple and easy to maintain but we argue that it is not suitable for a widely distributed environment such as the Grid. Firstly, it limits the scalability of the system as the number of users and the number of resources grows. A centralized system can in principle apply many optimization techniques such as using search trees, hash structures, indexing or caching data, for efficient data access. However, many criteria from the user are not simply concerned with the name but rather with a set of constraints, which may lead to the inappropriate use of these techniques. Secondly, dynamic information in the Grid such as CPU load, CPU idle, number of processes or free memory size might change rather fast. This will lead to the need to verify the validity of information on the server. The server can periodically poll the status from resources or a resource can update its information as its internal state changes. Again, we face with the scalability problem here. In addition, a resource might go down without any notification, making the information on the server uncertain. An alternative technique is to use a ”pull” model so that dynamic information is acquired directly from resources only when it is needed. Nevertheless, all of the above techniques are not sufficient in wide area distributed environments where the bandwidth might be low. Finally, with the centralized approach, the system is not fault tolerant. A crash on the server or a network-partitioning event might lead to the stop functionality of the system.

Distributed approach: distributed approach differs from centralized approach in the way the information is managed. Rather than storing information in a common place, information is distributed over the network. We argue that this approach is suitable to the resource discovery in widely distributed environments such as the Grid. First of all, distribution of information will increase the availability of the whole system. The failure of one information source will not interfere with the function of the others. Next, when a user performs resource discovery, it is reasonable to find resources near his place first (within his institution). It is only necessary to discover resources that do not belong to his institute when local resources do not satisfy the requirement. Therefore, putting information sources near resources can improve the overall performance of the system. Finally, because the information source is placed near resources, the distributed approach can handle dynamic information more efficiently based on the high-speed connections between resources.

We propose a fully distributed resource discovery for the ParoC++ runtime system. It differs from the centralized approach such as in Globus [26], NetSolve [15] or Condor [66] in the way the information is stored. Information about the resource in ParoC++ is fully distributed and is accessed on demand. This is somehow similar to P2P models [20, 68, 74] in the information distribution, but we add to our system the ability to self-adapt and to deal with dynamic information that does not exist in such P2P systems. Agent's and ours are comparable in the ability of learning but we use passive learning while agents use active learning by advertising themselves to the others periodically. The next section gives detailed information on our resource discovery model.

7.6.2. ParoC++ resource discovery model

7.6.2.1. Information organization

ParoC++ does not have any centralized server to manage resource information. Information about each resource is stored and accessed locally. Each information item is a pair of (type, value). In principle, the user can define any type of information he wants. However, in order for the ParoC++ resource discovery to work, some standard information types must be available. This information is listed in Table 7.1.

Information can be static or dynamic. Dynamic information can change fast, hence it is obtained on demand. Static information does not change during the life time of the service. Therefore, it is loaded from a configuration file when the service is started. In this configuration file, each line stores one information element, consisting of two text fields separated by a space: the type of information and the value.

Name	Value Type	Description
host	string	The full qualified host name of the local resource (static).
platform	string	The hardware architecture of the local resource (static).
maxjobs	integer	The maximum number of ParoC++ jobs (objects) can run simultaneously (static).
np	integer	Number of processors of the resource (in the case of SMP machine) (static).
jobmgr	string	The name of the job management system of the resource (static).
ram	integer	Maximum memory size, in MB (dynamic).
power	real number	Maximum available computing power for each processor of resource (dynamic).
jobs	integer	Number of jobs (objects) managed by ParoC++ that are currently running on the local resources (dynamic).

Table 7.1: Standard information types of resource

Information about the resource is queried via the protected method *Query* on the ParoC++ global service- a parallel object of type *JobMgr*. *JobMgr* inherits all global services (parallel objects) and it is a composite parallel object to access services.

```
virtual int Query(char *name, char *value);
```

If the information "*name*" is available, *Query* stores the corresponding value in the output argument "*value*" and returns true. Otherwise, *Query* returns false to the caller.

New dynamic information can be added to the existing system by just simply deriving a new class from *JobMgr* and by overwriting the method *Query* inside the new class. By this way, the user can also customize the calculated dynamic information such as providing a different benchmark method to fit the specification on his own resource.

7.6.2.2. Resource connectivity

Resource connectivity shows the knowledge of each resource about other resources inside the environment. In ParoC++, each resource is represented by a parallel object. We model the resource connectivity by a dynamic direct graph of objects. Edges of the graph represent the knowledge that one resource has over the others. Each resource learns about the others through the resource discovery process and hence it can add or remove edges to/from the graph.

In principle, the ParoC++ environment can be depicted as an arbitrary graph. However, the initial organizational structure of resources is recommended to be divided into two main levels:

- Local level: resources inside a single organization. At this level, some resources are chosen as the "masters" that manage the join of resources within the institute. A new resource joins the environment by registering itself to the masters.
- External level: the initial connectivity among institutes. The masters, also known as the "gateways" of one institute should know some resources (usually the masters) of some other neighbor institutes. This level tries to glue institutes to form a bigger connected graph of resources.

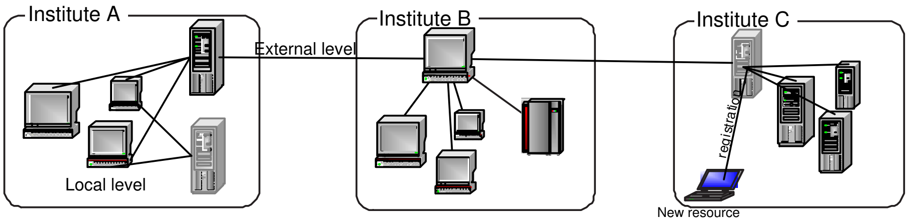


Figure 7.5: A recommended initial resource connectivity. During the resource discovery process, the master might not be necessary due to the learning of local resources.

Figure 7.5 shows an example of the initial resource connectivity graph consisting of three institutes A, B and C. Although resources within A do not know the resources of C but thank to the ParoC++ mechanisms, it is possible for the institute A to use the resources of institute C.

Inside each domain, the administrator should specify the initial connectivity among the resources within his domain as well as the outbound connectivity with other domains. The information obtained from the resource discovery process will make the system to be evolved and new connections will be automatically added and updated.

In order for a new resource to join the environment, it is necessary that the resource should initially know some others (usually the masters) inside the environment (vertices of the graph). The resource then registers itself to any nodes of the graph to create a new edge. Currently, this information is explicitly specified by the user through a configuration file when the service is started.

7.6.2.3. Resource discovery algorithm

The resource discovery algorithm of ParoC++ is based on three major steps: matching, forwarding and adapting. Upon receiving the resource request, each node (resource) will perform:

1. **Matching**: the OD is parsed and separated into several atomic items. Each atomic item contains a single requirement on a specific category of resource information (such

as computing power, memory available, etc.). The atomic item is then matched with the corresponding local information of resource. Matching is performed via the virtual protected method *Match* of the parallel object *JobMgr*:

```
virtual int Match(const char *requirements, float &fitness);
```

The method requires two inputs: the user requirements on the resource and the fitness (a non-negative number, fitness ≥ 1 means the resource is completely satisfied) showing the previous evaluation of the current requirements on other traversed resources.

If the resource satisfies the OD and the local fitness is bigger than the previously evaluated fitness, *Match* method returns successful. In this case, the resource discovery process then cancels the previous resource reservation, makes the local resource reservation, updates the fitness value. Otherwise, *Match* returns 0 and the fitness value is unchanged.

The evaluation of fitness value is based on the equation: $fitness = \min\{\frac{T_i}{R_i}\}$, where T_i and R_i are the total amount of resource currently available and the amount of resource required for the atomic item i in the requirement.

In addition to matching the requirement, authorizing the user for using the resource is also performed via the method *MatchUser*:

```
virtual bool MatchUser(const paroc_accesspoint &appservice);
```

MatchUser requires the access point to the application scope service (*appservice*) for which the resource discovery service can obtain necessary information about the user to perform the authorization. The default implementation of *MatchUser* always returns "true" (successful). The user will overwrite this method to implement himself the authorization mechanism.

If both matchings are successful and the fitness is greater than or equal 1, the resource discovery finishes successfully; the reservation ID and the local service access point are returned to the caller.

Otherwise, forwarding step is executed.

2. **Forwarding**: forward the request to the neighbor nodes by invoking the synchronous resource discovery methods. Information to forward includes:

    - The user requirements.
    - The current fitness value.
    - The current corresponding global service access point that partially satisfies the requirements.
    - The trace of nodes previously traversed (to avoid the forwarding loop).

Only neighbor nodes that are not listed in the trace will be considered for forwarding. The local node will add itself into the trace before forwarding. As the resources are likely organized in a tree-like structure, the number of resources that can be traversed is an exponent number to the depth of the tree (the size of the trace). So we will not need a big size of the trace. Currently, the maximum trace length is 32.

Forwarding process terminates successfully if a fully qualified resource is identified.

3. **Adapting**: Each node maintains a priority list of neighbor nodes that defines the order of forwarding. The newly discovered resource can also be added to this list. A node in the list can be static which means the user explicitly specifies or the neighbor node explicitly registers itself to this node; or it can be dynamic which means the neighbor node is added to the list by learning from the discovery. Static nodes are used to maintain the connected resource graph while dynamic nodes are used to improve the performance of the resource discovery process. We have three scenarios of node forwarding:

   - The resource is discovered by this forwarding. The factors used to calculate the priority of a node are: the time spending to find a resource and the fitness value of the found resource. We choose the priority H(A) of the neighbor node A as: $H(A) = \frac{old\_H(A)+\frac{T}{F}}{2}$ where the *old_H(A)* is the current priority of node A before forwarding, T is the time waiting for the result from this forwarding and F is the fitness value of this forwarding. The smaller value of H(A) means the higher priority of node A in the list. The position of A in the list is updated after this calculation. If the newly discovered resource is not in the list, it will be automatically added to the list.
   - The resource is not discovered by this forwarding. The new priority of the node is updated based on the time spent on this forwarding: $H(A) = old\_H(A) + T$
   - The forwarding fails due to the failure on the network or on the to-be-forwarded neighbor node: if the neighbor node is static, it will be temporarily removed from the list for a specific amount of time; otherwise, the node is permanently removed from the list.

   We also limit the maximum number of dynamic nodes in the list. When the number of dynamic nodes in the list passes this limit, the lowest priority dynamic node will be suspended from the list.

7.6.3. Access to the ParoC++ resource discovery service

The resource discovery service is accessible via the parallel object interface *JobMgr* using the virtual concurrent method:

```
virtual conc int AllocResource(paroc_accesspoint &appservice,
                         const char *objname, const char *OD,
                         char *codefile, paroc_accesspoint &jobcontact);
```

To find a resource for a parallel object, the user needs to provide three inputs: the application scope service access point of the application (*appservice*) for the ParoC++ code manager service, the name of parallel object (*objname*) and the evaluated object description string (*OD*). If the resource which satisfies the OD is identified, *AllocResource* stores the ParoC++ object manager access point (see section 7.7) of the discovered resource in *jobcontact*, the corresponding object code location in *codefile* and returns the reservation identifier (a positive number) of the parallel object allocation request on the discovered resource. All of these outputs are used by the ParoC++ object manager to launch the parallel object. If no such a resource is available, *AllocResource* returns -1.

7.7. ParoC++ object manager

The ParoC++ object manager is used as a portal to access the local resource. It provides two main functionalities: launching the parallel object and managing the resources.

7.7.1. Launching the parallel object

As a suitable resource for a parallel object is discovered through the resource discovery service, the next action is to launch the object using the object manager service on that resource. Launching a parallel object consists of the following steps:

1. Locate the corresponding binary code of the parallel object and download the code if necessary. The code file can be stored in the local file system or it can be put on a remote server that is accessible via some well-known protocols by any resource inside the computational environment. Currently, ParoC++ supports HTTP and FTP protocols.

2. Setup the call back service. The call back service is a temporary network service to receive the access point of the to-be-created parallel object.

3. Submit the object code to the local job manager of the resource. The submission is performed in a separate process. Information passed by the object manager during the submission includes:

    - The global service access point. ParoC++ provides two options: the global service access point can be the same as the one at the interface part of the ParoC++ application or it can be the access point of the resource where the object is to be created.

- The application scope service access point. This should be the same for all parallel objects of the same ParoC++ application.
- The parallel object name. Since each code file may contain the code for several parallel objects, so the object name is necessary for the ParoC++ runtime system to choose the proper parallel object code to start.
- The call back access point so that the parallel object access point, upon created, can be transferred back to the object manager.

ParoC++ is designed to deal with the heterogeneity. It allows a job to submit to different "local" job management systems such as LSF [88], PBS [45] or even Globus [24] (Section 7.10 describes the Globus integration into ParoC++). ParoC++ is a high level infrastructure. Rather than providing its own authentication/authorization services, ParoC++ is based on the security infrastructure of the underlying resource management system.

4. Wait on the call back service for the access point from the parallel object.

Launching a parallel object is performed through the virtual method *ExecObj* of the global service interface *JobMgr*:

```
virtual int JobMgr::ExecObj(const char *codefile, const char *objname,
         int reserveId, const paroc_accesspoint &globalservice,
         const paroc_accesspoint &appservice, paroc_accesspoint &objcontact);
```

The interface should provide the object name (*objname*), the code location (*codefile*), the reservation ID on the discovered resource (*reserveId*), the global service access point (*globalservice*) and the application scope service access point (*appservice*). *ExecObj* will execute the object server and return the execution status. If the parallel object is successfully started, the access point to that object will be stored in the output argument *objcontact*. This access point is used later by the interface to establish the connection to the object server.

7.7.2. Resource monitor

The resource monitor is internally used to manage the state of resource during the object allocation and the object execution. It consists of:

- Manage the resource reservation for parallel objects. During the resource discovery process, as soon as a better fit resource is discovered, the reservation is moved from the old resource to the new one.
- Monitor the termination of parallel objects. When a running object terminates, the resource monitor should free immediately all resources occupied by the object so that other objects can be allocated.

7.8. Parallel object creation

The interaction between ParoC++ services is illustrated in Fig. 7.6. In the figure, the number on each edge shows the order of tasks which will be performed to create a parallel object. Each parallel object comprises two parts: the interface part from which the user creates the object and the server part that can run remotely on a different resource. Manipulations on the parallel object from outside are handled by the ParoC++ compiler via the object interface. The creation of an object is managed by the ParoC++ system and is transparent to the programmer. This process consists of the following steps:

1. Create the parallel object interface.
2. The interface evaluates the object description (OD) and calls the ParoC++ resource discovery service to find a suitable resource.
3. The resource discovery service running on the resource checks if the local resource satisfies the OD and the corresponding object code is available for the local platform.
4. If the local resource does not fit the OD, the request is then forwarded to other neighbor nodes.
5. As a suitable resource is identified, the access point of the object manager is returned to the interface
6. The interface connects to the object manager on the newly discovered resource and invokes *ExecObj* to launch the object server.
7. The object manager can also access the application scope service to performed extra tasks such as obtaining user identity for authentication and authorization. These extra tasks are customizable by overriding the suitable methods of the *JobMgr* parallel object.
8. If the object code is stored on a web server, the object manager downloads it.
9. Setup the temporary call back service and execute the object code on the local resource.
10. The object server creates an access point and transmits it back to the object manager via the call back service. The object server then starts waiting and serving invocation requests.
11. The object access point is returned to the interface.
12. Thank to the access point, the interface creates a TCP/IP connection to the object server. It then transfers all data under the Sun XDR format through this connection to the object server and perform the object construction (constructor invocation). All invocations on the object afterward will also go through this connection directly to the object server.

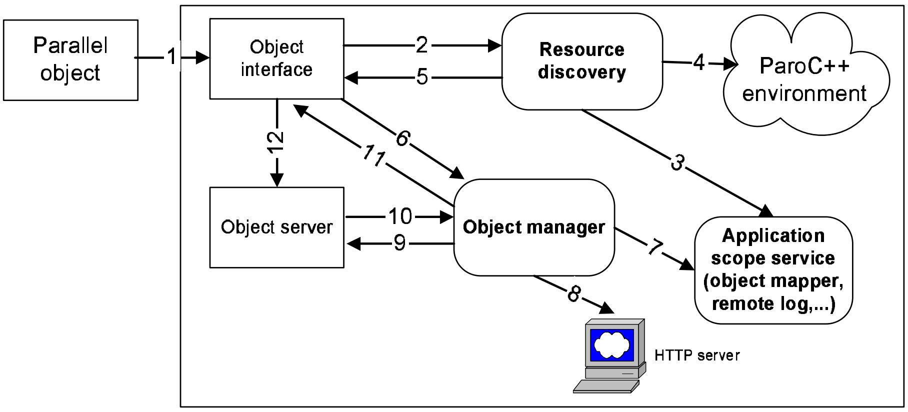


Figure 7.6: Parallel object creation process

7.9. Fault tolerance of the ParoC++ services

Application scope services are application dependent, so any failure on these services does not affect other applications. Therefore we will focus on the fault-tolerance of global services of ParoC++: the resource discovery service and the object manager service.

In ParoC++, a failure can come from the network or an internal error on the machine where the ParoC++ service is running. We describe here how the ParoC++ services handle these two types of failures.

All ParoC++ services are implemented as parallel objects, so we depend on ParoC++ exception mechanisms to detect system failures. Any system failure on object invocations will raise an exception of type *paroc_exception*.

7.9.1. Fault tolerance on the resource discovery

ParoC++ follows the fully distributed approach for resource discovery. The resource topology is represented as an arbitrary dynamic graph where the vertices stand for resources and the edges stand for the knowledge of one resource over its ”neighbor” resources. No centralized server is required to perform the resource discovery. Each node rather plays an equal role inside the system.

The failure can occur on forwarding the resource discovery request from one node to a neighbor node. If the neighbor is static, i.e. the neighbor is explicitly specified by the user, the edge is temporarily removed from the graph. Because static nodes maintain the connected property of the resource graph, so upon the occurrence of failures on a neighbor, the node will pause the forwarding on this neighbor for a period of time before re-adding the neighbor to its neighbor list. Otherwise, the edge is permanently removed from the graph.

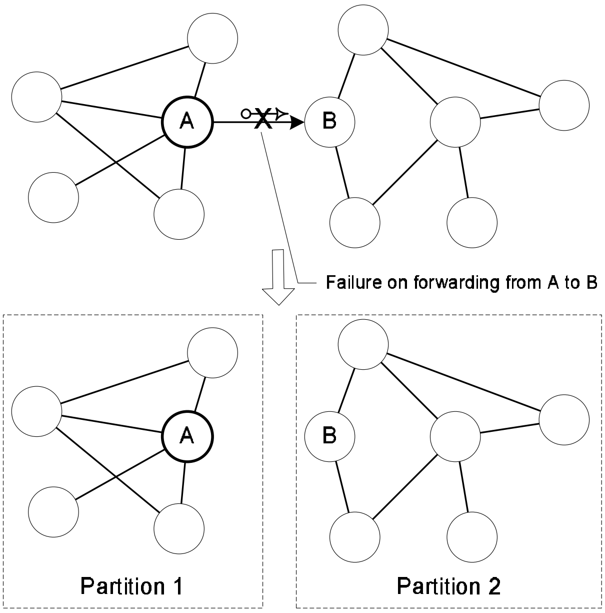


Figure 7.7: Resource graph partitioning due to failures

The failure can lead to the graph partitioning as illustrated in Fig. 7.7. In the figure, we assume that there is no other edges but $< A, B >$ connecting the left and right sub-graphs. The failure on $< A, B >$ will cause network partitioning. How the ParoC++ system reunifies when the failure is recovered? There are two possibilities:

Case 1 The edge $< A, B >$ is initialized by B through the self-registration of B to A when B is started. In ParoC++ system, a node will periodically register itself to its static neighbors. Hence, either the failure of A or B or the network partition is recovered, the edge will be automatically re-initialized.

Case 2 The edge $< A, B >$ is initialized explicitly by A when A is started. If A is down, when it is recovered, the edge will be automatically initialized. If B or the network connection between A and B is down, B is temporarily removed from the neighbor list of A. A will periodically try to add B to its neighbor list until it can do that successfully.

During the resource discovery process, some temporary resource reservations can take place and these reservations will be canceled when a better matched resource for the object is discovered. Failures can occur during the cancellation. Such failures do not affect the integrity of the system and are handled by aging each reservation inside the object manager service.

7.9.2. Fault tolerance on the object manager service

The object manager service is directly used by the object interface during the object creation. Failures related to this service will prevent the parallel object server from starting correctly.

It is mainly caused by the three following reasons:

1. The parallel object server side does not start successfully on the local resource due to an internal fault (such as out of memory, maximum number of processes reached or fail to download the object code). This fault leads to the time out error on the object manager service on waiting for the parallel object access point. The local resource is temporarily out-of-service (the matching process in the resource discovery service of the local node will return false) for a specific amount of time. The object manager will return an error code to the interface.

2. The object server side fails to start because the resource reservation has been canceled due to the time out event. An error code is also returned to the interface.

3. The service is unreachable. In this case an exception will be thrown on the interface.

At the interface side, when an error on the object manager service occurs, it retries the object creation process from the beginning (see section 7.8).

7.10. ParoC++ as a glue of Grid toolkits

In the ParoC++ service architecture, we focus on the abstraction of the *ParoC++ service semantics*: what the outcome of the service is, not how the service is implemented. ParoC++ services are represented as abstract parallel object classes. Users will implement the services on their own systems, either directly or implicitly via some low-level toolkits, by deriving new classes from these abstract classes. By doing this, we try to hide the implementation heterogeneity of the whole system behind the abstract service classes.

From the abstract level, ParoC++ applications require from the execution environment two basic functionalities:

- Find and reserve a resource that satisfies the requirements (OD).

- Execute a program (object) on a specific resource.

Depending on the toolkit, there are two possible levels of integration of the Grid toolkits into ParoC++: the complete integration or the partial integration. In the complete integration, the user needs to completely implement the ParoC++ global service interfaces using the low-level functionalities of his toolkit. Many practical toolkits do not provide enough functionality required by ParoC++ such as requirement-based resource discovery or advance resource reservation. In order to use these toolkits in the ParoC++ system, the partial integration allows the user to re-implement only portions of the service architecture that are well-supported by the toolkit while still keeps the overall architecture of the system. For example, we can use the security service of the low-level system to authorize the user as an extra task on launching a parallel object.

Depend on the toolkit, the user may also need to extend the application scope services to provide additional functionalities that are required by the new global service implementations. For instance, in order to identify the user, the application scope service should be extended to provide the user certificate which requires by the new global services.

7.10.1. Globus toolkit integration

We describe an example of the integration of the Globus toolkit [28, 30] into ParoC++. Currently, the Globus toolkit version 3 (GT3) does not provide an automatic resource discovery mechanism based on the resource requirements yet, so we will try the partial integration approach: we keep the resource discovery architecture but we additionally implement the security support for ParoC++ object allocation using the Globus security infrastructure [85] and the Globus Resource Allocation Manager (GRAM) [24] for executing the parallel object server.

7.10.1.1. Application scope service for Globus

Globus is currently based on X.509 certificates [80] for authentication and authorization users. The private key for the certificate is stored on the user's local machine and will not be transmitted over the network. Therefore, any functionality that uses the Globus security infrastructure should be performed locally. The new application scope service for Globus extends the abstract application scope service with the following functionalities:

- Provide (but not check) the user identity (the Globus's subject name).
- Submit a job on a remote Globus node. We use the Globus GRAM API [37] to submit a parallel object to a Globus node.
- Monitor the status of the Globus submitted jobs.

All of these functionalities can not be performed on remote resources since they require the user's private key for the authentication.

7.10.1.2. Resource discovery service for Globus

We keep the same resource discovery architecture as described in section 7.6. For each Globus node in the resource graph, we extend the resource discovery service by overwriting the user matching method:

```
virtual bool MatchUser(const paroc_accesspoint &appservice);
```

MatchUser will obtain the user's subject name from the Globus implementation of the application scope services and look for the subject name in the list of authorized subject names of the local Globus node from the Globus map file. *MatchUser* is successful if the subject name exists in the list. *MatchUser* does not change any thing in the local system, so no security method is required.

7.10.1.3. Object manager service for Globus

This is the most complicated part of the integration. We overwrite the two virtual protected methods:

1. *ExecObj*: execute the object server. This is done through the newly created interface of the application scope service for Globus. If the parallel object is successfully started, the Globus job ID is returned. We will store this ID as well as the access point to the application scope service to monitor the object status later.
2. *Update*: check and update the parallel object execution status. Each time a resource discovery request is received, the object manager updates the resource information first by calling *Update* method before performing matching. For the Globus implementation, the status of each running parallel object is checked by accessing the corresponding Globus application scope service to acquire the status from this service.

7.10.1.4. Interaction of Globus-based ParoC++ services

Figure 7.8 shows how the ParoC++ system interfaces with the Globus toolkit for a parallel object allocation. This is similar to the general object creation scheme in Fig. 7.6 that we have presented in section 7.8.

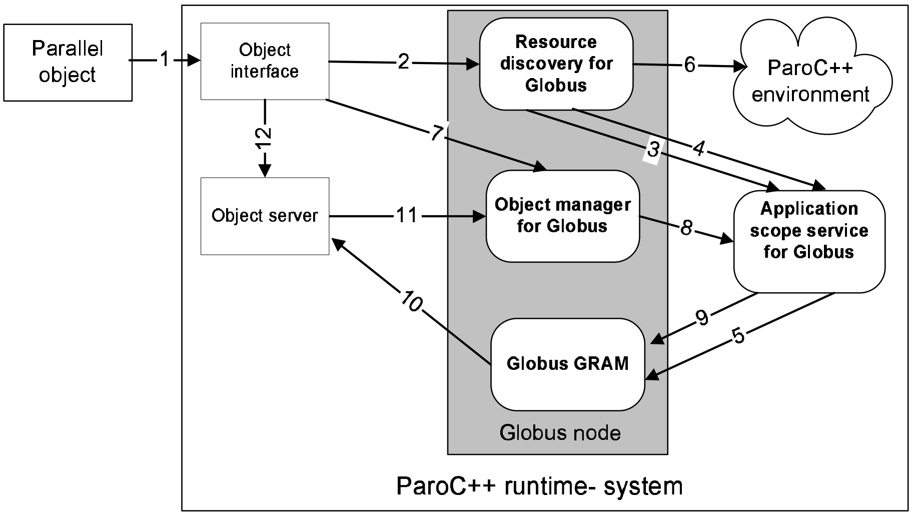


Figure 7.8: Interaction of Globus-based ParoC++ services during a parallel object creation

The object creation process consists of the following steps:

1. Creation of the parallel object interface.

2. The interface accesses the resource discovery service for Globus.
3. The resource discovery acquires the Globus user name (subject name) from the application scope service and checks if the Globus user is authorized to use the local resource.
4. The resource discovery updates the information about Globus jobs by accessing the Globus application scope service.
5. The application scope service accesses the Globus GRAM to obtain the necessary job information.
6. If the local resource does not satisfy the requirement or the authorization fails, the request is forwarded to other ParoC++ nodes.
7. When a suitable resource is identified, suppose that the resource is managed by the Globus, the interface invokes the virtual method *ExecObj* on the object manager service, which has been customized for the Globus, to launch the object.
8. The object manager has the Globus application scope service submit the Globus job on behalf of the user who has started the application.
9. The application scope service access the Globus GRAM to submit the job.
10. The Globus GRAM actually executes the parallel object.
11. The parallel object access point is transferred to the object manager which is, in turn, returned to the interface.
12. The interface connects to the server and the creation process is completed.

7.11. Summary

This chapter completes the discussion of the chain user-application-infrastructure started from chapter 5 with the ParoC++ runtime infrastructure. We define two types of services in this infrastructure: the application scope services and the global services. Application scope services are bound to the application to serve application specific tasks. Different applications will have independent instances of the same application scope services. Two application scope services: *code manager service* to manage multiple object code files of the ParoC++ application and *remote console service* to collect and print out messages from remote distributed components to the local application console are described. Global services provide access to the infrastructure. *Resource discovery* to locate a resource with a required performance and *object manager* to remotely execute parallel objects and to monitor the object execution are two essential services for running ParoC++ application. One important aspect of the ParoC++ runtime architecture is the inter-operability between the global services and the application scope services to perform application related tasks. The fault tolerant issue of these services is also discussed.

We design ParoC++ services as the high level abstractions that separate the service interfaces from the service implementations. Each service is represented as a parallel object with a well defined object interface. The user can provide different implementations for different platforms or he can overwrite some functionalities (methods) by deriving a new service object from the existing one. Representing services as parallel objects makes the services more customizable and extensible to different Grid environments. The last part of the chapter discussed about how to use the ParoC++ system to access other Grid computing environments with an example of the Globus toolkit integration. By this way, the ParoC++ runtime system can be used as the glue of different Grid environments.

Chapter 8

ParoC++ for solving problems with time constraints

8.1. The Framework

We built on top of ParoC++ a framework for developing time constrained applications using the parallelization scheme presented in chapter 4. The framework provides users a tool to represent their problems and their decomposed sub-problems, the dependencies of sub-problems, the complexities of problems and the time constraint within which the whole problem must be solved. To solve the problem, the framework will perform the following tasks automatically and transparently to the user: estimate the time constraints of sub-problems; compute the resource requirements; instantiate a suitable solution based on the currently available resources; and schedule the problems to be executed. This framework frees users from the complexities of the execution environment.

The framework is developed based on the two main object abstractions: the sequential ”node” object that represents a node in the decomposition tree and the parallel object ”problem” that represents a sequential solution to the problem or the sub-problem at the corresponding node. These two types of abstractions co-exist and co-operate to the execution of the application. Sequential objects are used as the skeleton for constructing nodes of the decomposition tree and the decomposition dependency graphs in the parallelization scheme. It is also responsible for creating the problem's parallel object. The programmer will construct the parallelization scheme by deriving new classes from these abstractions.

8.2. Expressing time constrained problem

The skeleton, illustrated in Fig. 8.1 consists of two main classes: `DTreeNode` and `ProbObj`. `DTreeNode` is a sequential class representing a node (a problem or sub-problem) of the decomposition tree (DT). Each `DTreeNode` object associates with at most one parallel object of type `ProbObj` which implements the problem solving on a remote resource.

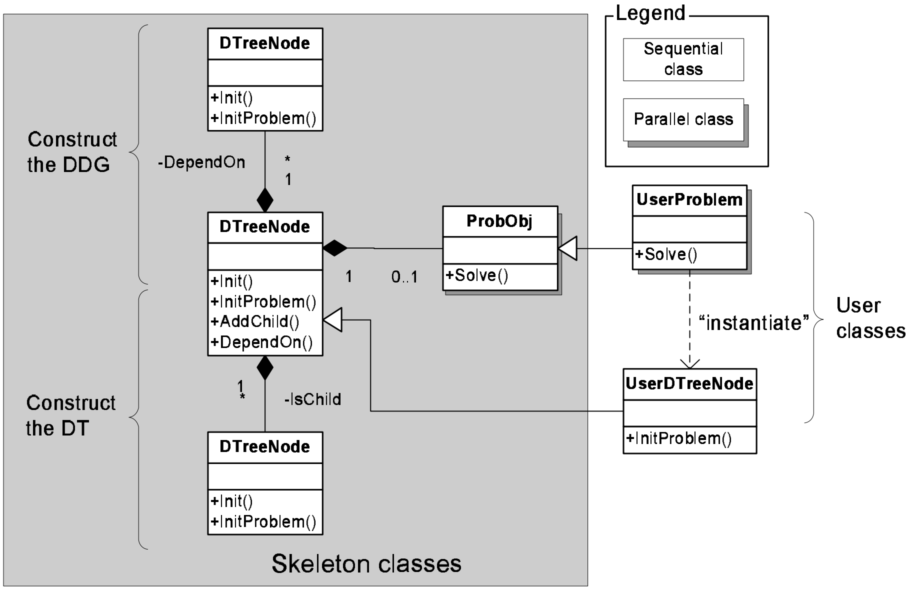


Figure 8.1: The UML class diagram of the framework

The user gets his problem solved by following the four steps bellow:

- Create the parallelization scheme.
- Set the time constraint.
- Instantiate the solution.
- Execute the parallelization scheme.

The rest of this section will explain these steps.

8.2.1. Creating the parallelization scheme

The framework allows the user to construct the parallelization scheme which consists of a decomposition tree (DT) and a set of decomposition dependency graphs (DDGs). The DT and DDG can be constructed locally on the machine where the user starts the application.

First, the user needs to derive new sequential "node" classes from `DTreeNode`. Each node class represents a type of nodes in the user's DT and is associated with a parallel class derived from `ProbObj`. The parallel class implements the actual problem solving of the node. Inside each node class, the user should overwrite the skeleton `InitProblem` method. `InitProblem` tries to create a corresponding parallel object of the problem (or sub-problem) with a given computing power requirement, which has been automatically computed by the skeleton. Any failure on the parallel object creation (out of resource) in `InitProblem` leads to the "NULL" value returned which tells the framework to perform the "decompose" step.

Secondly, the user should construct node objects for each node of the DT from the node classes. On constructing a node object, in addition to the initialization data for the node, the user also needs to provide the complexity of the underlying problem on that node. This complexity will be used by the framework to compute the time constraints for all DT nodes later on.

As all DT nodes have been constructed, the next step is to connect these nodes together to form the parent-child edges by calling the method `AddChild` on the parent node of DT:

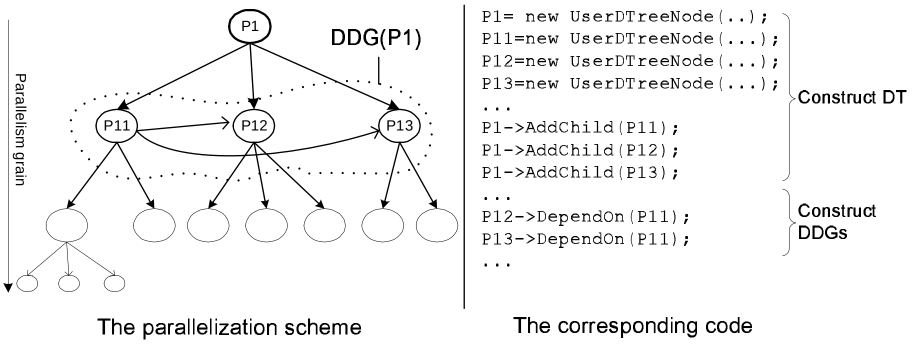
```
void DTreeNode::AddChild(DTreeNode *child);
```

The user can control *the constraint guard coefficient* (see section 4.4.2) for each decomposition step from a parent to the child nodes by invoking the method `SetCoeff` on the parent node of DT:

```
void DTreeNode::SetCoeff(float coeff);
```

Finally, the user needs to define the decomposition dependency graphs (DDGs)- dependencies between direct child nodes of the same parent by invoking the method `DependOn`:

```
void DTreeNode::DependOn(DTreeNode *prior);
```



```
P1= new UserDTreeNode(..);
P11=new UserDTreeNode(...);
P12=new UserDTreeNode(...);
P13=new UserDTreeNode(...);
...
P1->AddChild(P11);
P1->AddChild(P12);
P1->AddChild(P13);
...
P12->DependOn(P11);
P13->DependOn(P11);
...
```

The parallelization scheme

The corresponding code

Figure 8.2: Example of constructing a parallelization scheme using the framework

Figure 8.2 shows a parallelization scheme and a sample code to construct this scheme: `UserDTreeNode` is a user-defined sequential class derived from `DTreeNode`. We illustrate the code for only one decomposition step from the "root" problem `P1` to 3 sub-problems `P11`, `P12` and `P13`. After constructing the nodes, we call the method `AddChild` to construct the DT and finally, the method `DependOn` to construct the DDG(P1).

8.2.2. Setting up the time constraint

After the parallelization scheme has been constructed, the user can set the time constraint for his problem by invoking the `SetTimeConstraint` method on the root of the DT:

```
void DTreeNode::SetTimeConstraint(float time);
```

in which `time` is the number of seconds that the user expects the problem to be solved.

For all non-root nodes, the time constraints, when necessary, will be automatically computed based on the time constraint of the root, the decomposition dependency graphs and the complexities given by the user upon constructing the DT nodes.

8.2.3. Instantiating the solution

This step is to instantiate a suitable solution to the problem on the Grid by just performing a simple call to `DTreeNode::Init` on the root of the DT.

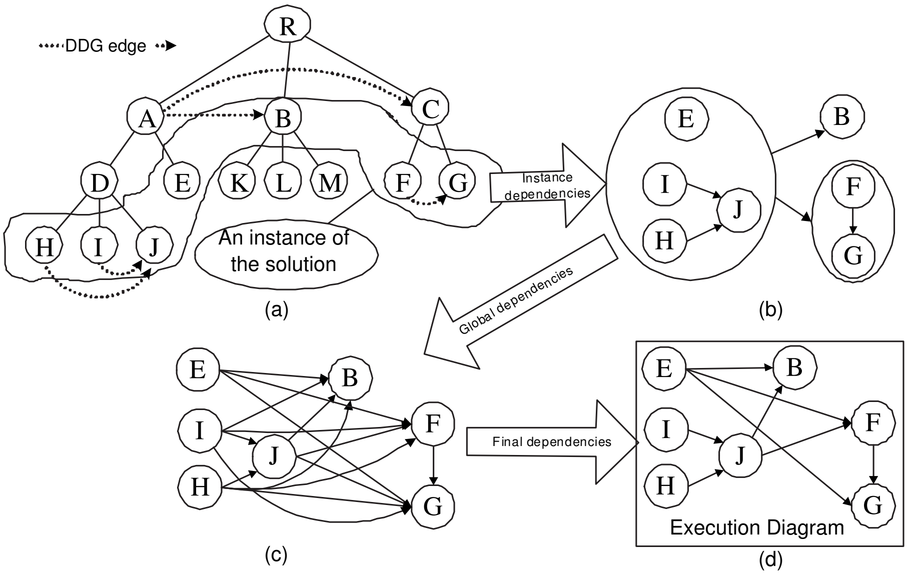


Figure 8.3: Initializing the parallelization scheme

`DTreeNode::Init` will do the following tasks:

- Find an instance of the solution with respect to the availability of resources in the computational environment (Fig. 8.3.a). As we have described in section 4.3.2 of chapter 4, a try-and-decompose process will be performed on the DT starting from the root. "Try" on a node (method `InitProblem`) will compute the resource requirements regarding the time constraint and the complexity of the node. It then tries to allocate a parallel object with the computed resource requirements. The ParoC++ runtime system will perform the resource discovery and the resource matching. If "try" succeeds, the initialization for the parallel object is called. If not (the resource is not available),"decompose" will be executed. First, "Decompose" evaluates the time constraints for all child nodes based on the DDG, the time constraint of the parent node and the complexities of all child nodes. Then, the try-and-decompose is performed again on each child node. Decompose fails if the child node is a leaf. In this case, `DTreeNode::Init` will return "out of resource". Otherwise, in the end, an instance of the solution is identified.

- Find the global dependencies of problems within the instance of the solution. When an instance of the solution is found, `DTreeNode::Init` will construct the global dependencies of problems within that instance by merging all hierarchical DDGs (Fig. 8.3.b) to generate a unique dependency graph of all problems (Fig. 8.3.c). All redundant dependencies will be removed from this graph to generate the final dependency graph: the execution diagram (Fig. 8.3.d).

- Elaborate the execution diagram to each problem's parallel object. Each ProbObj-based parallel object (problem) contains: a counter that counts the number of problems that this problem directly depends on; and a set of parallel objects that will be executed next as this problem is completely solved. For example, in Fig. 8.3.d, the counter value of problem "F" is 2 since F depends on "J" and "E"; and the "next" parallel object of "F" is "G". DTreeNode::Init updates this information based on the execution diagram.

8.2.4. Executing the parallelization scheme

The last step is to call the DTreeNode::Solve method on the root of the DT to get the problem solved. DTreeNode::Solve looks for all "ready" nodes (nodes with no coming edge) in the execution diagram and then asynchronously invokes ProbObj::Exec on all ready nodes. Each time a problem (ProbObj-based parallel object) finishes, it will "fire" all next problems in its list (by invoking the remote concurrent method Exec). A problem, when being fired (waiting for a "FIRE" event from its event sub-system), will check its counter. The counter value of 0 means all previous problems have been solved. In this case, it will start solving by invoking its local virtual method ProbObj::Solve (the user needs to overwrite this method). Otherwise, the counter is decreased and the parallel object waits for the next "being fired". The execution process is similar to that of a neural network.

8.3. Elaborate the skeleton to the user's problem

The user implements his problem by deriving two classes from DTreeNode (sequential class) and ProbObj (parallel object class-or parallel class for short). Each DTreeNode-based class should be associated with a parallel class. The user's program should look like:

```
//Declaration part of sequential objects and parallel objects
class MyDTreeNode : public DTreeNode
{
  public:
    MyDTreeNode(..., float complexity);
    virtual ProbObj *InitProblem(float mflops);
};

parclass MyProbObj: public ProbObj //problem's parallel object
{
  public:
    MyProbObj(..., float mflops) @{ power=mflops;...other OD items here... }
  private:
    virtual void Solve();
 };
 //Implementation part
 MyDTreeNode::MyDTreeNode(...,float complexity):DTreeNode(complexity)
 {
 ....
 }
 ProbObj *MyDTreeNode::Initproblem(float mflops)
 {
    MyProbObj *prob;
    try
    {   prob=new MyProbObj(...,mflops);
    }
    catch (paroc_exception *e)
    {   printf("Creation of parallel object fails\n");
        return NULL
    }
    //do additional initialization of MyProbObj here....
    return prob;
 }

 MyProbObj::MyProbObj(...,float mflops)
 {
    //do initialization here..
 }

void MyProbObj::Solve() {
 //The user implements how to solve his problem here
}
```

The user overwrites two virtual methods: InitProblem (in the DTreeNode-based sequential class) to create his own problem (parallel object) and Solve (in the ProbObj-based parallel class) to implement how the problem is remotely solved. Inside InitProblem, the user performs the ”try” step by creating the problem's parallel object (MyProbObj). If no resource with the required power is available, thank to the exception mechanism of ParoC++, a system exception of type paroc_exception will be automatically thrown. The user will ”catch” this exception in order to know if ”try” step is successful or not. InitProblem then returns to the framework a pointer to the newly created parallel object (try succeeds) or ”NULL” (try fails). The latter case means the problem can not be solved sequentially and therefore, the ”compose” step will be automatically performed by the framework.

Next, the user needs to build the parallelization scheme. The user creates MyDTreeNode objects and connects these objects together by using AddChild and DependOn.

Finally, he needs to call: `SetTimeConstraint` to set the time constraint for the original problem and to have the framework automatically compute the time constraints for sub-problems; `Init` to instantiate a suitable solution based on the available resources and to compute the execution diagram for that configuration (see section 8.2); and `Solve` on the root node (`MyDTreeNode`) of the DT to actually start solving the problem.

8.4. Summary

The chapter completes the parallelization scheme that we described in chapter 4 with a ParoC++ framework for solving time constrained problems on the Grid. This framework provides users a tool to express: the problem and decomposed sub-problems, the dependencies of sub-problems in each decomposition. The idea is to represent the user's problems as the abstractions of parallel objects and the parallelization scheme as the abstractions of sequential objects. The user will provide his own implementations of problem and sub-problem solutions by deriving new parallel objects from the existing ones inside the framework. The dependencies of sub-problems within a decomposition step should be explicitly specified by the user. The framework will automatically compute the time constraints of sub-problems and instantiate a suitable solution depending on the availability and the characteristics of resources inside the Grid environment. By this way, the user can concentrate on the "logic" of the problems rather than taking into account the complexities of the executing environment.

Some experiments on using this framework have been performed. The results will be presented in chapter 9 and chapter 12.

Part III

Experiments

Chapter 9

Experiments

9.1. Introduction

We present in this chapter various experiments on the ParoC++ system and the parallelization scheme framework. First, we test the communication cost of ParoC++ method invocations and make the comparison with that of MPICH on the same hardware. The second one is to use ParoC++ to compute big matrix multiplication with passive data access (chapter 6). We will demonstrate in this test how ParoC++ with different data access methods is used to achieve the performance. The first two tests are performed in a homogeneous environment of Pentium 4, 1.7 GHz, Linux network of workstations with 100Mbit fast Ethernet connections.

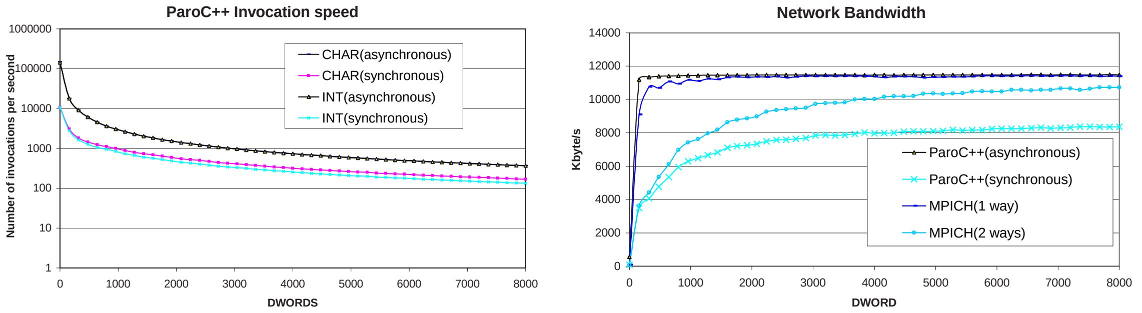
The last test is running on an environment of sparc/Solaris and Pentium/Linux machines where we try to emulate the heterogeneity of the Grid and to illustrate how the time constraint problems can be solved in such environments.

9.2. ParoC++ benchmark: communication cost

We wrote a program containing two objects called "Ping" and "Pong" running on two different machines. Ping invokes methods of Pong with different arguments (size and type) and with two different invocation semantics: synchronous and asynchronous. Invocation speed, invocation latency and the communication bandwidth are measured. We then compare the results with a similar ping-pong program written in the de facto standard implementation of Message Passing Interface-MPICH [42, 43].

Figure 9.1(a) shows the invocation speed of objects on 8-bit and 32-bit integer messages. Asynchronous invocations are more efficient than synchronous ones, especially for small messages due to the message aggregation. The average latency of asynchronous invocation is rather low, about 6.9 μsec because of the possible overlap between invocations (MPICH: 43 μsec) and of synchronous one is about 94 μsec (MPICH: 123 μsec).

The communication bandwidth of ParoC++ method invocations is then compared to



(a) (b)

Figure 9.1: Parallel object communication cost

the MPICH implementation. Figure 9.1(b) shows that asynchronous invocations, due to the overlap, utilize better bandwidth than synchronous invocations. This bandwidth is slightly better than the asynchronous send (one way) of MPICH. The bandwidth of asynchronous calls almost reaches the limit of the Fast Ethernet throughput (11.3 MB/s). For synchronous invocation, MPICH achieves somehow better bandwidth in our experiment (15-20% better for large messages). This is due to the extra cost for marshalling data and multiplexing remote methods in ParoC++ that can not be overlapped at both interface and server sides. Although ParoC++ is less efficient than MPICH in synchronous communication operations but it provides much higher abstraction based on objects than MPICH. Moreover, in parallel computing, overlapping between computation and communication is important to achieve the performance. Therefore, asynchronous communication should be used whenever it is possible.

9.3. Matrix multiplication

We have performed a test with passive data access (chapter 6) on a matrix multiplication. The matrix sizes used in the test are 5000x5000 and 4000x4000. The parallel algorithm for multiplying matrices AxB consists of a data object that stores all matrices data (A, B and the output matrix). The algorithm is divided into two phases: initialization phase: matrix B will be distributed to all workers (Solver objects); and computation phase: Solver objects will acquire some rows of A from the data object, perform the multiplication with matrix B, and send back the results to the data object. In each Solver, there are two buffers, size of 20 A-rows each, and a local copy of B. When a buffer is used up, a request for new data for that buffer is invoked on the data object while the computation will continue on the other buffer. The data object, upon receiving the request, will send A-rows to the corresponding Solver. This processing is repeated until all rows in A have been sent out to Solvers.

We also compare the results with a similar algorithm using MPICH. In the MPICH-based version, the master distributes data to workers and receives back the results from workers. Workers play the role of Solvers. The only difference is that in the MPICH-based version,

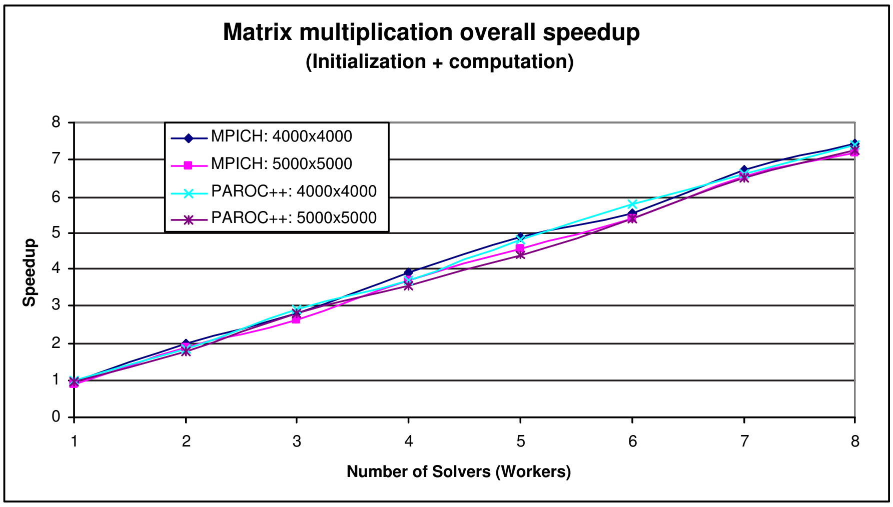


Figure 9.2: Matrix multiplication speed up on Linux/Pentium 4 machines

each worker has only one buffer to store rows of A from the master for the computation. The reason of using only one buffer in the MPICH version is that it is harder and probably unsafe to control the concurrency inside each MPICH process as required by the passive data access.

Figure 9.2 shows the speedup results of the algorithm in compare to the sequential algorithm. The graph shows that with both MPICH and ParoC++, we can achieve almost linear speedup up to 8 processors. Since the computation time is much bigger than the time for distributing matrices to solvers (workers), the speedups with 1 processor are 0.97 (matrix 4000x4000) and 0.95 (matrix 5000x5000) in the ParoC++ version; 0.95 (matrix 4000x4000) and 0.88 (matrix 5000x5000) in the MPICH version. With 8 processors, the speedups are almost the same in both versions (ParoC++ and MPICH): 7.4 (92.5% efficiency) for 4000x4000 matrix size and 7.2 (90 % efficiency) for 5000x5000 matrix size. Although ParoC++ programming abstraction is much higher than that of message passing, ParoC++ can achieve a comparable performance with MPICH in the test.

The overall performance illustrated in Fig. 9.2 can be decomposed into two parts: the initialization part in which one matrix will be sent to all Solvers (one-to-one sending via method invocations in the case of ParoC++ and broadcasting in the case of MPICH); and the computation part in which the matrix multiplication on each Solver (worker) is performed.

Figure 9.3 shows the time spent for distributing one matrix to all Solvers (workers). The initialization time in the case of ParoC++ is almost linear to the number of Solvers due to the sequential property of the method invocation while the MPICH broadcast primitives is more efficient since it uses a tree-like broadcasting algorithm. We could also implement a similar algorithm but for simplicity, we did not implement it in this experiment.

The second part that contributes to the overall speedup in Fig. 9.2 is the computation time described in Fig. 9.4. As the number of Solvers (workers) increases, the computation

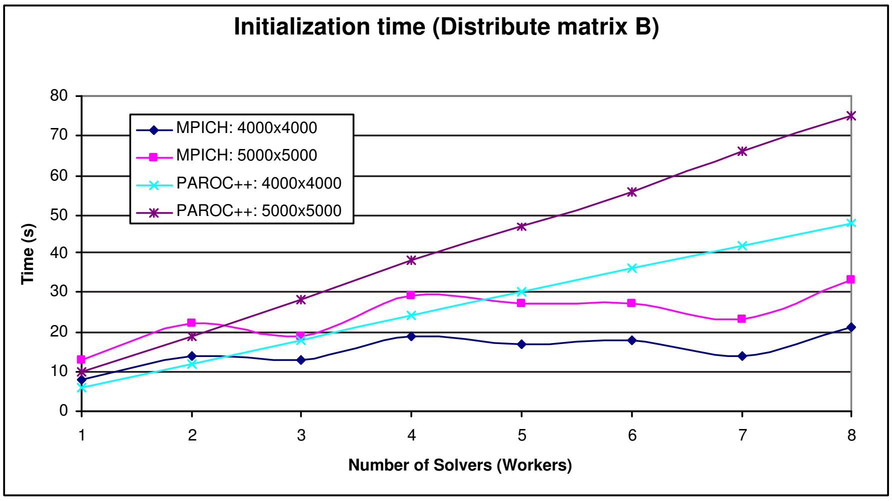


Figure 9.3: Initialization part: distributing of one matrix to all other Solvers (workers)

time in the ParoC++ version is smaller than that of the MPICH version. That is because the ParoC++ version uses double buffers of A-rows with passive data access which improves the overlap between computation and communication and hence can enhance the performance. With 8 processors, the computation time in the ParoC++ version is 5-7% faster than that in the MPICH version on the same hardware architecture.

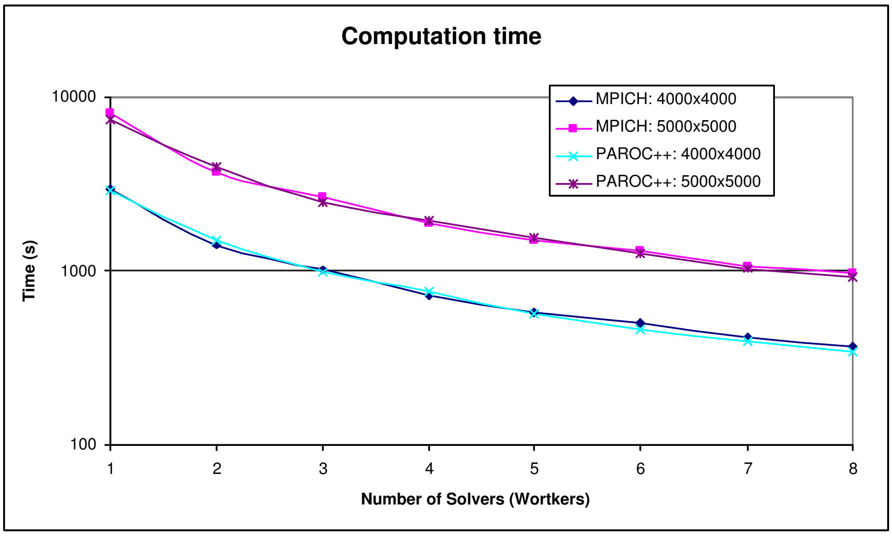


Figure 9.4: Computation part: each Solver (worker) will request for A-rows from the data source (master) and performs the multiplication

The experiment shows that passive data access can be easily used in ParoC++ to prepare data in advance and hence to enhance the overlap between communication and computation for high performance.

9.4. Time constraints in a Grid-emulated environment

The last test in this chapter is to demonstrate how ParoC++ with the parallelization scheme is used to solve time constrained problems. We first build a Grid emulated environment with 130 machines. A parallelization scheme with different time constraints is then constructed and executed in that emulated environment.

9.4.1. Emulating Grid environments

For the experiment, we built an environment with the following characteristics:

- High heterogeneity in computing power of processors
- Different hardware architectures
- Different operating systems
- Different network topologies

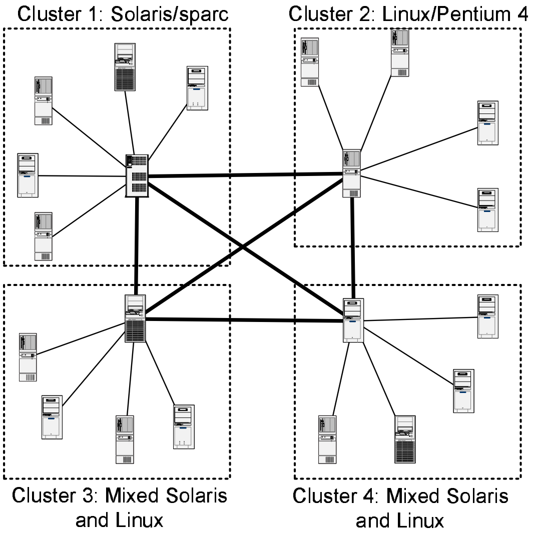

Cluster 1: Solaris/sparc
Cluster 2: Linux/Pentium 4
Cluster 3: Mixed Solaris and Linux
Cluster 4: Mixed Solaris and Linux

Figure 9.5: Initial topology of the environment

The environment consists of 130 machines running Linux and Solaris, divided into 4 heterogeneous clusters. The network topology is described in Fig. 9.5. Each cluster has one "master" machine that controls the job submissions for that cluster (resource discovery and resource allocation). Initially, one master has no knowledge about machines in other clusters except the other masters. The effective computing powers of nodes in the environment are emulated, ranging from 60MFlops up to 1GFlops (Fig. 9.6). Some machines are SMPs with 2 or 4 processors. Each node runs the ParoC++ services which are required for the resource discovery and the parallel object execution.

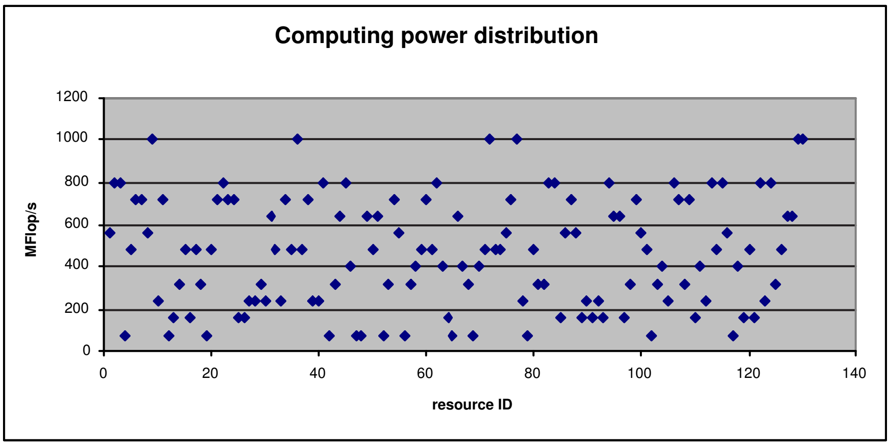


Figure 9.6: Distribution of computing power of heterogeneous resources

By building such an environment, we want to somehow simulate the wide area distributed environment such as the Grid: each cluster can be considered as one organization where we know all resources. However, we usually do not know in detail about resources of other organization except the "gateway" (the master machine). The resource allocation should be performed within the "local" organization first. Only when no resource is locally available, then the request will be forwarded to its "neighbor" organizations via the gateways. The ParoC++ job service also learns new resources from the resource discovery results.

9.4.2. Building the parallelization scheme

We assume that the problem to-be-solved requires 50GFlop (total number of floating point operations). We also assume that for each problem , we can decompose it into 4 sub-problems (the degree of DT) and the depth of the DT is 4. Hence, our DT contains 341 nodes (problems or sub-problems).

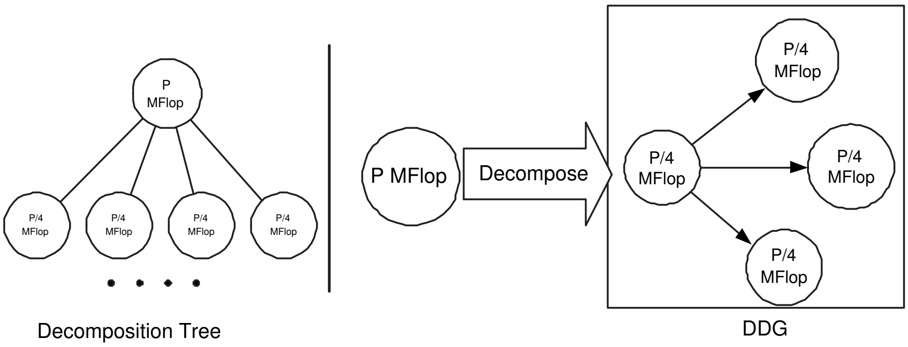


Figure 9.7: Decomposition Dependency Graph for each decomposition step

In order to classify the decompositions based on the dependencies of sub-problems, we introduce a new metric called *Parallel factor*. Parallel factor is defined as *the ratio between the* *maximum number of problems that can be solved in parallel and the total number of problems.* The parallel factor is a number in between 0 and 1. The parallel factor is 1 if all problems within a decomposition step are independent and can be solved in parallel; is 0 if all problems are dependent and they must be solved sequentially one after the other.

In the first test, all sub-problems within a decomposition step are independent and each sub-problem requires 1/4 computing power of its parent. The parallel factor is 1. In many practical problems, dependencies are inevitable due to the nature of the decomposition. Such dependencies will degrade the degree of parallelism. So, in the second test, we create a DDG for each decomposition step as in Fig. 9.7: in each decomposition step, 25% of the operations is spent to solve one sub-problem sequentially and after that three other sub-problems can be solved in parallel. In this case, the parallel factor is 75%.

We constructed 3 classes: `MyDTreeNode` (sequential, from DTreeNode), `MyProbObj` (parallel, from ProbObj) and `LogDataObj` (parallel class used to log execution progress information). The first two classes are used to construct the parallelization scheme. `LogDataObj` is a shared parallel object among all `MyProbObj` object. Each `MyProbObj` object will invoke methods on the shared `LogDataObj` object to store information about its execution states. In many real applications, `LogDataObj` can be replaced by the data source, the output or the monitoring parallel objects. The `MyProbObj` object will simulate the real computation by a counting loop. The time for running the loop depends on the computing power of the resource and the complexity (total computing power) of the object (this information is obtained from the parallelization scheme).

9.4.3. Time constraints vs. execution time

We run the tests on the built heterogeneous environment with different time constraints. The selected *constraint guard coefficient* (see section 4.4.2) for each node is 0.95. The real computation time is measured and compared with the time constraint. For each run, the number of parallel objects that reflects the degree of parallelism is also counted.

Figure 9.8 shows the experiment results. When the time constraint is greater than or equal to 50 seconds, the problem is solved sequentially because there exists some 1GFlops machines. The actual solving time in this case depends on the resource discovery process but it is always smaller than the required time. As the required time decreases, the problem starts to be decomposed (number of parallel objects increases). For tests with the parallel factor of 100% (there is no dependency of sub-problems), the problem can be solved with the time constraint of 2 seconds (67 sub-problems). Below that, the problem can not be solved due to the lack of resources. In the case where the parallel factor is 75% (see 9.4.2), the dependencies reduce the capacity of parallelism and increase the demand for resources. Therefore, we got ”out of resource” message when the time constraint is less than 10 seconds. The time constraint of 10 sec leads to a decomposition of 64 sub-problems and the actual

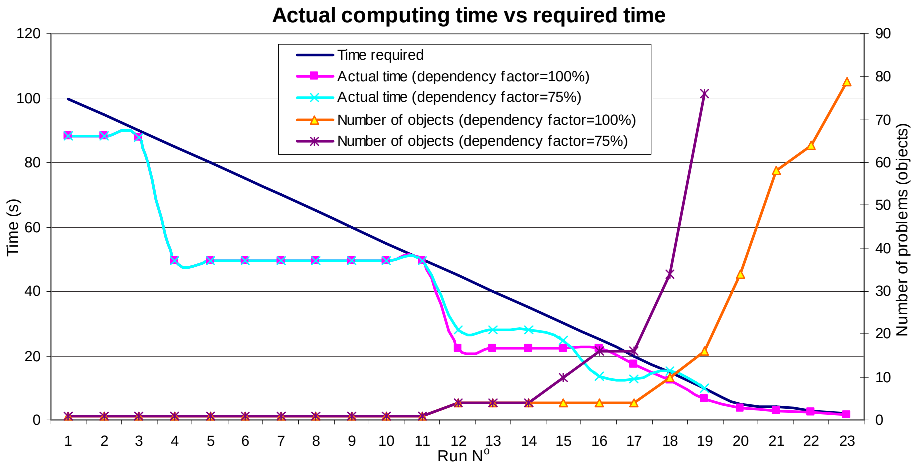


Figure 9.8: Emulation results with different time constraints

running time is 9.67 sec. Nevertheless, in both cases, the time constraints have been achieved.

9.5. Summary

The chapter focuses on two main points: at the programming level, we demonstrate different programming aspects of the ParoC++ system; and at the concept level, we illustrate how to express and to solve the time constrained problems using the parallelization scheme framework on the Grid.

At the programming level, ParoC++ has been used to measure the communication cost of method invocations and to do a parallel matrix multiplication. The results are then compared with those of MPICH on the same hardware and same compilation options. The two tests confirm that ParoC++ can achieve a similar performance in comparing with the de-facto standard message passing library MPICH. Passive data access can be easily applied to ParoC++ programs to improve the data movement between different distributed components.

To demonstrate how to use the parallelization scheme framework, we have presented our experiment on a Grid emulated environment. This experiment mainly takes into account the computing power of problems. The network bandwidth and the communication latency are not yet mentioned although the user can control these parameters by providing a suitable parallel efficiency for each node of the decomposition tree and by adding the network specification to the OD. The experiment running on the emulated environment shows how the time constraint be satisfied by deriving automatically different grains of parallelism driven by the available resources. In other words, we illustrate how to tailor an application to the Grid environment. The experiments on 130 mixed Linux/Solaris machines show that through the parallelization scheme framework, the time constraint goal has been achieved. Chapter 12 will present how to solve a real problem using the parallelization framework on these 130 machines.

Chapter 10

Test case 1: Pattern and defect detection system

10.1. System overview

This test case is an industrial application of image processing for tissue manufacturing in the framework of the European project Forall[1]. Pattern and defect detection system (PDDS) is a part of the whole chain of the Forall system as illustrated in Fig. 10.1. The Artificial Vision subsystem is integrated in the whole management and control architecture of the Forall system. Its goal is to collect images of textile in an appropriate way and to transmit them to the PDDS system which runs remotely on a different machine. PDDS will analyze the images to find pattern positions and to detect defects on these patterns. The output of PDDS will be the input for the Nesting system to cut the tissue in order to minimize the wasted textile material.

The analysis of images of PDDS should follow the real-time speed of the conveyor of at least 3.3MPixel/s.

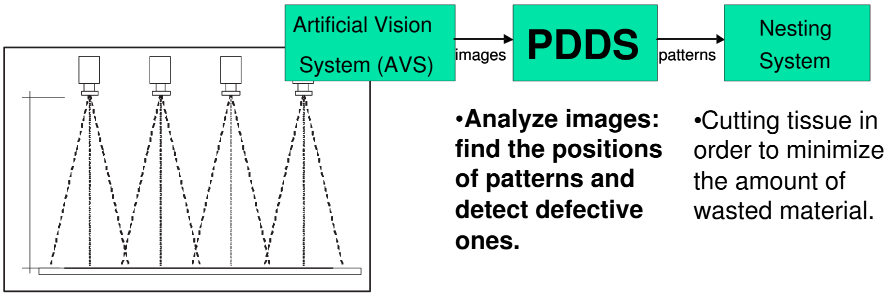

Artificial Vision System (AVS)
images
PDDS
patterns
Nesting System
•Analyze images: find the positions of patterns and detect defective ones.
•Cutting tissue in order to minimize the amount of wasted material.

Figure 10.1: Overview of the Forall system for tissue manufacturing

[1] FORALL-"Parallelization and optimal implementation of compute-intensive tasks for industrial applications", projet EUREKA -Nr.: E!1955 / CTI 5130.1

10.2. The algorithms

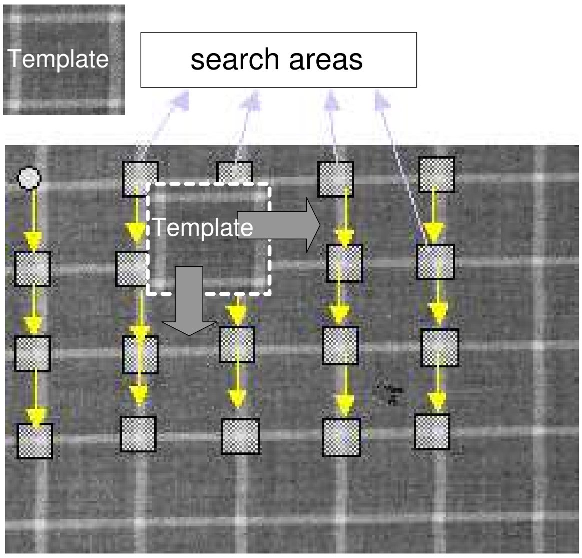


Figure 10.2: PDDS algorithm

The idea around the PDDS algorithms is to search all over the tissue the local maximal values of similarity between the user-provided pattern template and the sub-image. Such positions are considered as the start points of patterns. PDDS optimizes the algorithm by searching only in small areas (the highlight areas in Fig. 10.2) for the patterns on the next row. The similarity criterion used in PDDS is cross-correlation which can be efficiently calculated by the mean of fast Fourier transform. More details on these algorithms can be found in [53, 60].

10.3. The parallelization

The parallelization follows the classical master-slaver model in which a master will split images into smaller sub-images and send these sub-images to other slavers. The slaver will do computation and return the results to the master.

Figure 10.3 demonstrates the parallel object diagram of PDDS using ParoC++. The main program will create two parallel objects of type `ImageBuf` and `OuputData` and several parallel objects of type `Analyzer`. `ImageBuf` and `OutputData` objects are shared among `Analyzer` objects. The `Analyzer` objects access the `ImageBuf` object to get the images (synchronous invocation), analyze them and then store the results in the `OutputData` object (asynchronous invocation). `ImageBuf` is responsible for receiving image frames from the image acquiring system (AVS), splitting them into small images and storing these small images into

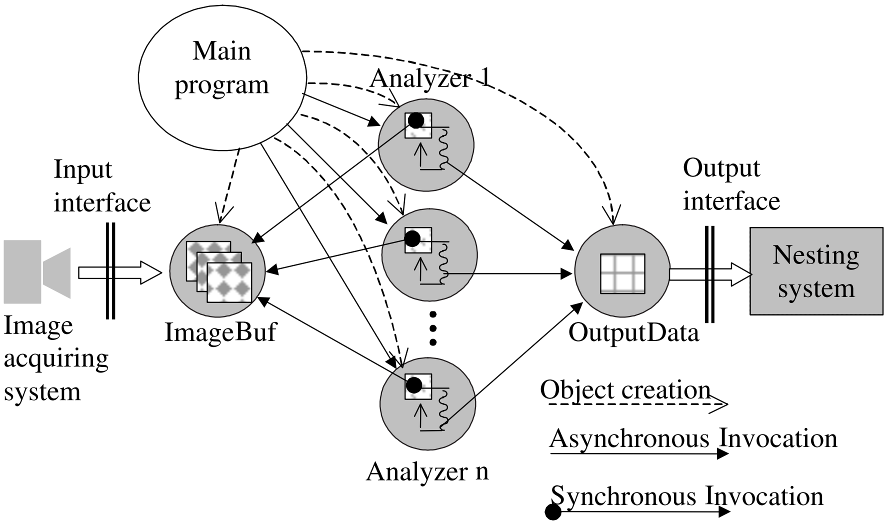


Figure 10.3: ParoC++ implementation of PDDS

an internal buffer so that the `Analyzer` objects can get and analyze. The main program also plays the role of a monitoring agent. It watches over the `ImageBuf` to check if the system could follow the real-time speed. In the case the main program detects that the system overworks due to the increase on the computation demand or the external changes on the resource loads, it can create some more `Analyzer` objects to speed up the computation. Hence, in PDDS we also deal with the adaptation of the application to the user requirement and to the dynamic state of the environment.

We will do the experiment with two modes of data access: passive data access (see chapter 6) where `ImageBuf` will stores sub-images directly to a Solver's image buffer upon requested and active data access where each Solver will actively get a sub-image from `ImageBuf`.

10.4. Experiment results

10.4.1. Computation speed

The input for the first experiment consists of 100 frames and is sent to PDDS frame by frame. Each frame has the size of 2048x2048 pixels (8bit-gray images, 4MBytes/frame). `ImageBuf` will split the frame into several sub images of the size 512x512 pixels. Neither the adaptation to the environment nor the adaptation to the changes in user's requirements is considered in this test. Figure 10.4 shows the speedup of two types of tissues: small patterns (Sict2) and big patterns (Monti) on a network of Sun sparc workstations and on a cluster of Pentium 4 with the active data access mode. We see that in both environments, almost linear speedup is achieved. PDDS runs about 14 times faster on 16 processors.

We then do the same experiment as the above but we use the passive data access mode to

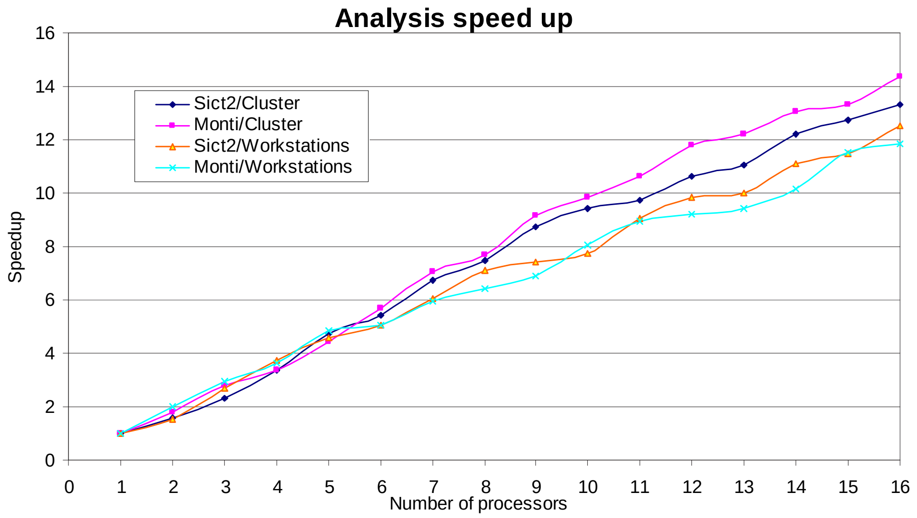


Figure 10.4: Speed up of PDDS implemented using ParoC++ with active data access mode

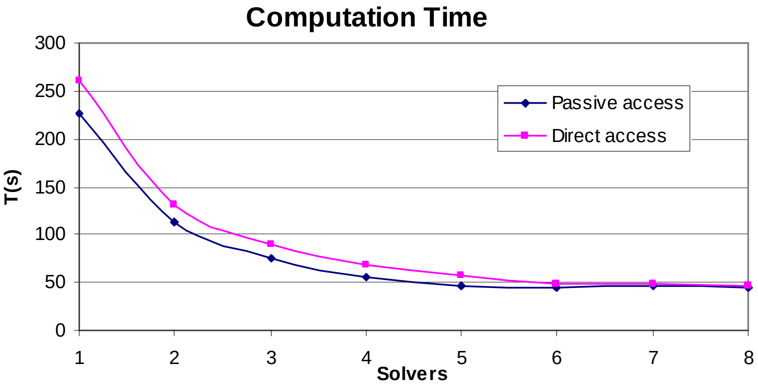


Figure 10.5: Passive access vs. direct access in PDDS

acquire the sub-images. The comparison of the two data access mode is shown in Fig. 10.5. Passive data access gives better results than direct data access (7- 15% more efficient). The time for sending all images to the network takes about 42 seconds over the total computation time of 44.95 sec (8 processors, passive access). That means almost all computation on the `Analyzer`s has been overlapped with the communication.

10.4.2. Adaptation

The second experiment illustrates the adaptation capacity of the ParoC++ application to external changes. Two types of changes are considered: the changes on the requirement from the user and the changes from the environment. Inside the data source (`ImageBuf`), the analysis speed is measured as the rate of image frames sent out to the `Analyzer` objects. The "main" program, after creating parallel objects will poll the data source periodically to acquire information about the analysis speed. This information is then compared to the required speed from the user. When the real speed is smaller than the required speed, the main program will assume that there are some changes from the environment (external changes) or from the user (internal changes). In this case, it creates a new parallel object and connects this object to the data source. Then the polling process in the main program continues.

In the experiment, we will emulate both external and internal changes: increase the required analysis speed every two minutes and restart a machine where an Analyzer object is running (failure emulation).

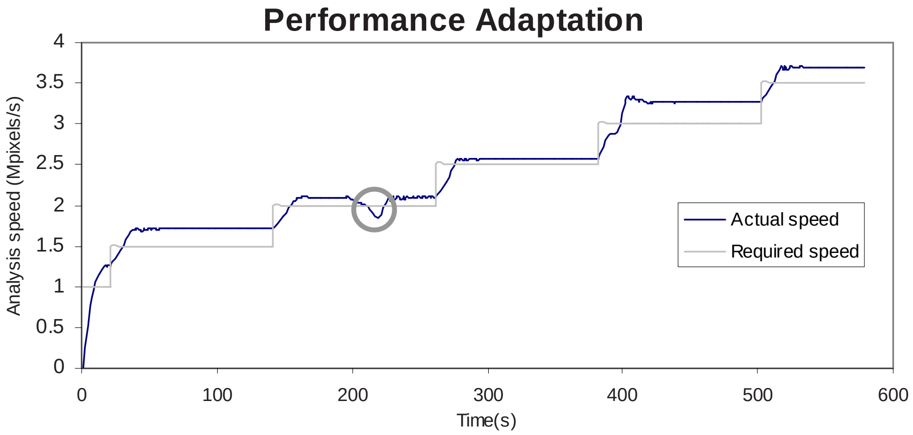


Figure 10.6: Adaptation to the external changes

The PDDS runs in a heterogeneous environment of Solaris/sparc and Linux/Intel with the adaptation part turned on. In figure 10.6, we show the dependency between the analysis speed (in term of MPixel/s) and the time. The gray line presents the required power whereas the bold line is the actual performance of PDDS. In the test, we dynamically change the requirement speed every 2 minutes. Due to these external changes, additional Analyzer objects (resources) are automatically allocated in order to satisfy the required performance. One interesting note is that at the second of 220, the actual performance goes down. The reason is that we have restarted a machine used by PDDS. The system reacts to this change by allocation more objects and is soon recovered to the normal speed. By this experiment we want to show the two important points:

- ParoC++ application can efficiently deal with the computation on demand.
- ParoC++ can adaptively use the heterogeneous resources efficiently.

10.5. Summary

The chapter presents the first test case of ParoC++ on an image processing application. ParoC++ has been used to develop the Pattern and Defect Detection System (PDDS) used in textile manufacturing. We have illustrated the object-oriented programming aspects in ParoC++, the high performance design and the capacity of the application to adapt to the dynamics from the environment and the user. The test case also exhibits an efficient data access method of ParoC++ and the feasibility of building an industrial application from scratch.

The next chapter will present another test case where ParoC++ is used to parallelize and to integrate an existing software.

Chapter 11

Test case 2: Snow modeling, runoff and avalanche warning

11.1. Introduction

Snow development and transformation is a complex natural process. Figure 11.1 illustrates different factors that are involved in the snow development: the wind causes snow erosion, the radiation of the sun brings energy that makes snow to be transformed (snow metamorphism), the vegetation also affects the energy absorption of snow and the mass balance, etc.

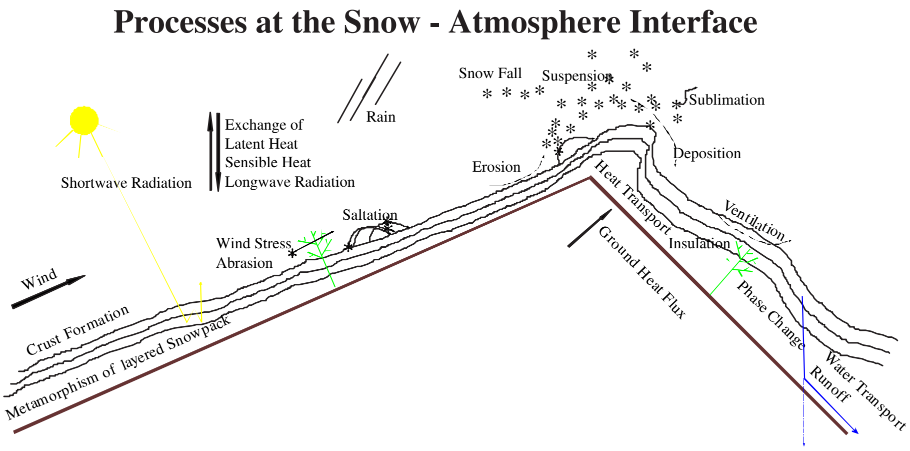


Figure 11.1: A complex system of snow development(source: M. Lehning et al., SLF-Davos)

The snow modeling, run off and avalanche warning system for the Alpine region of Switzerland called Alpine3D has been studied and constructed at the Swiss Federal Institute for Snow and Avalanche Research (SLF) in Davos, Switzerland. At the current state, Alpine3D consists of three coupled models as described in Fig. 11.2: the flow model (SnowDrift) to simulate the drifting snow; the snow model (SnowPack) to simulate snow fall and snow formation;

and the energy balance model (EnergyBalance) to calculate the radiation factors that will be absorbed by the snow. All of these models share the geographical information (topography, land use) about the region to be studied (Alpine region). Other models (e.g. wind model, vegetation model, etc.) can be integrated to the Alpine3D in future. Further information about the physical snow process models can be found in [6, 56, 55].

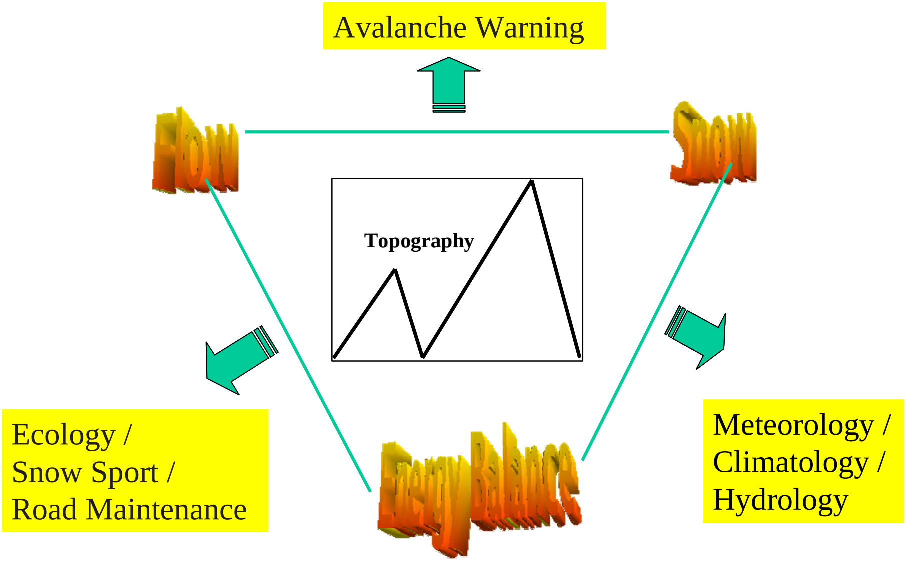


Figure 11.2: Model coupling for studying snow formation and avalanche warning

Alpine3D is a data and compute intensive application. Experiments on an UltraSparc/Solaris machine show that a sequential run of the Alpine3D takes about 5 hours CPU time to compute the snow simulation in 120 hours in the Gaudergrat area (Switzerland) 6.1x6.1 km^2 (with snow drift and run off generated) at the rather high resolution of 100m. Parallelization is necessary to make the simulation become feasible in a larger region or at a higher resolution.

We are working with SLF/Davos in the Hydro@Alpine3D project specifically aimed at parallelizing the software and deploying it on the Grid environment. ParoC++ has been chosen as the tool to perform this task.

Alpine3D is an existing application developed by different people in different domains: physicist, meteorologist, environmentalist, etc. Different languages have been used to develop the software: C, C++ and Fortran. Communication between modules was through files and was off-line. These create many difficulties for scientists on building a complex system where all components should work together and exchange data every computation step. Therefore, ParoC++ is not only the tool to parallelize the application for performance but it is also used as the tool to "glue" the differences of modules for building a highly integrated complex system.

While in the test case 1 in chapter 10, we have presented how to use ParoC++ for developing high performance applications from scratch, this chapter will show that ParoC++ can also be used to parallelize an existing application.

11.2. Overall structure of Alpine3D

Before ParoC++ came into the scene, the Alpine3D consists of three independent modules: SnowPack, SnowDrift and EnergyBalance. Communication between these modules are off-line and through files. Snowpack and SnowDrift was written in C and EnergyBalance was written in C++ and Fortran. All of these modules are still in the developing phase where new things are continuously integrated into the system.

ParoC++ has been used to couple these modules to form a unique application on the Grid and parallelize inside each module. This process is done in parallel with the development of algorithms and the evolution of the software modules.

The overall structure of Alpine3D with ParoC++ is shown in Fig. 11.3. The Alpine3D system consists of two main parts: the I/O part and the computation part. The I/O part contains 3 components: the *Input* to provide information about the topography, initial snow cover conditions, meteorological conditions, etc; the *Simulation parameters* to provide geophysical parameters of the simulation and the *Output* of the simulation. The computation part consists of three computation modules (*SnowPack*, *SnowDrift* and *EnergyBalance*) driven by the *Simulation control* module. The *Simulation control* module can turn on or off a computation module depending on the simulation parameters from the I/O part, control the simulation time and synthesize the results and output them to the I/O part. The tree computation modules will exchange data in each simulation step.

11.3. Parallelization of the software

In this section we will present two phases of the parallelization process in Alpine3D using ParoC++: first we will couple the three computation modules so that pipeline processing can be applied (coarse grain parallelism) and finally, we will try to parallelize inside the computation modules thus implementing a finer grain of parallelism.

Using ParoC++ brings many advantages to the Alpine3D system. First, the user doses not need to transfer the input data to remote machine before doing the computation. Input data can be stored locally on the user machine and be accessed through the parallel object interface on demand. The output of the simulation can also be transparently synthesized and stored back to the user machine. Secondly, the user can have a high level of modularity of the software so that the parallelization process can be performed independently inside different modules. High level abstractions from the object-oriented design allows the user to customize each functionality such as different model of snow development, energy balancing, etc. while still maintaining the global structure of the system. The object-oriented design also allows

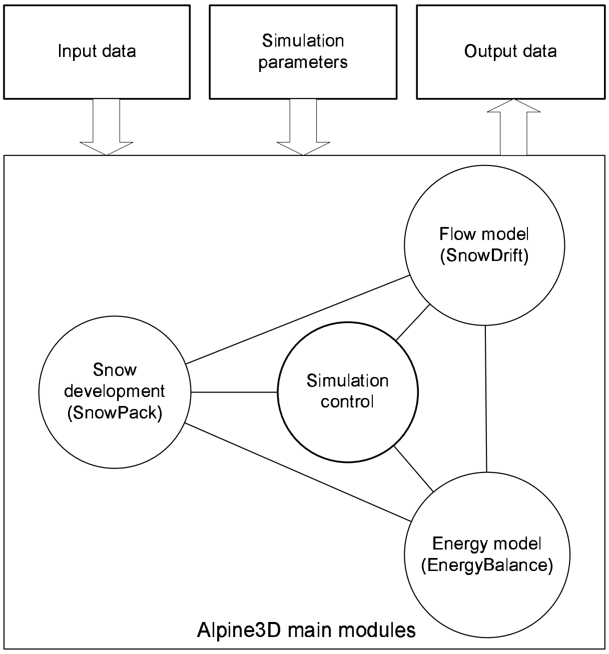


Figure 11.3: The overall architecture of Alpine3D

the user to reuse the existing code developed for the sequential version so the development of the software can take place concurrently. Finally, testing and debugging the program is difficult on the Grid. ParoC++ allows the user to test the semantics of the program using pseudo parallel objects just by replacing parallel objects by sequential objects and run the program as if it is a sequential one. This can facilitate largely the debugging and testing process.

Figure 11.4 shows the class diagram of the parallel version of Alpine3D. We focus on an open design of the application so that other modules can be integrated easily into the system. The input and the output of Alpine3D are designed as two parallel objects `InputObj` and `OutputObj` which can take data from files (`InputFile`, `OutputFile`) or from other modules. Depending on the simulation parameters controlled by `AlpineControl`, the SnowDrift module can be switched on or off. Currently, we have different input and output data sets whether the SnowDrift is on or off. Such differences are customized in the derived classes from `InputFile` (`InputFileDrift`, `InputFileNoDrift`) and `OutputFile` (`OutputFileDrift`, `OutputFileNoDrift`).

`InputObj` and `OutputObj` are shared among modules (`AlpineControl`, `SnowDrift`, `SnowPack`). Each module will acquire the inputs directly from the `InputObj` and output the results via `OutputObj`. Only computed data will be exchanged between modules.

The gray region in Fig. 11.4 shows the core of module coupling which will be presented in section 11.3.1. The two parallel classes `SnowDriftMaster` and `SnowDriftWorker` implement the parallelization of SnowDrift computation which will be presented in section 11.3.2.

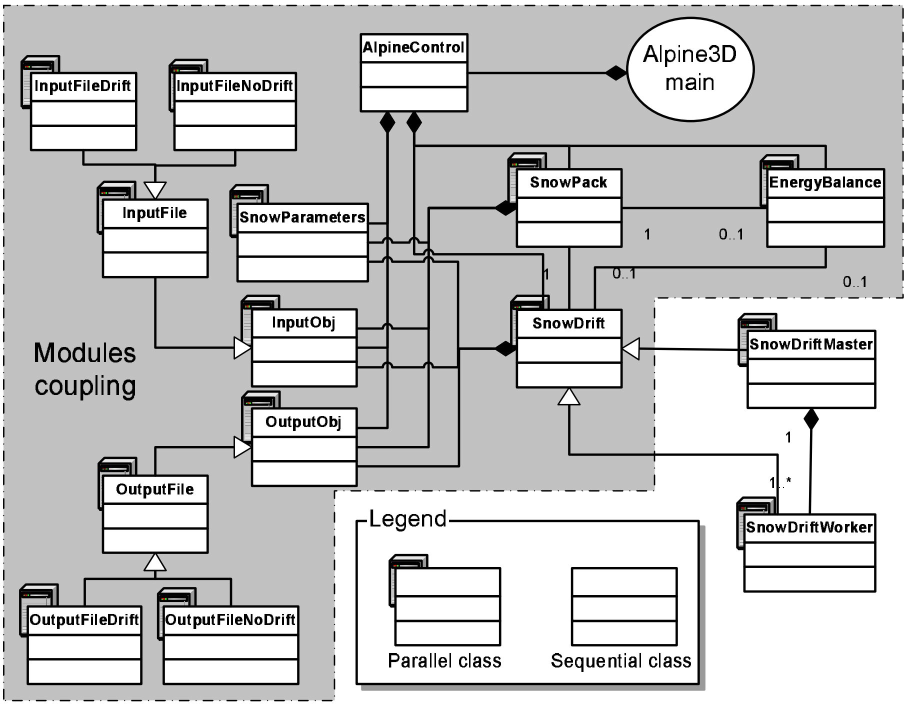


Figure 11.4: UML class diagram of parallel and sequential objects in the parallel version of Alpine3D

11.3.1. First part: Coupling modules

Our first task is to couple the three computation modules SnowPack, SnowDrift and EnergyBalance so that they can run in parallel on the Grid. During each simulation step (time unit), each module will interact with the others. Figure 11.5 shows the data flow between these three modules.

SnowDrift acquires the meteorological data, the wind field from the input module, performs the computation on the data (3D grid) to produce required 2D meteorological information and passes them to SnowPack and EnergyBalance. It then computes snow saltation and suspension (3D grid) and derives the snow mass change (2D grid) on the snow surface. This information will be passed to Snowpack.

EnergyBalance will perform the calculation of the sun rise and the sun set time for a specific Julian[1] day. After that, it will calculate the radiation components absorbed for each 2D grid point on the snow surface. It takes into account shadowing and reflection through topography. This information is then passed to SnowPack.

SnowPack will compute the snow accumulation over time based on the weather condition provided by SnowDrift and the radiation components from EnergyBalance. Information on snow surface (2D grid) is then updated on SnowDrift and EnergyBalance for the next simulation step.

[1] The order of a day in the year

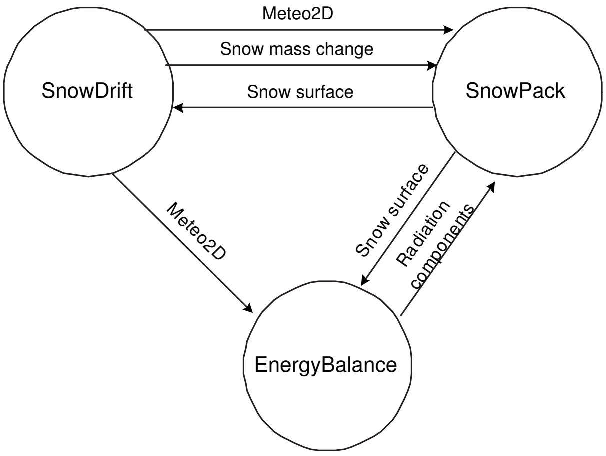


Figure 11.5: The data flow between SnowPack, SnowDrift and EnergyBalance during a simulation time step

We use the ParoC++ parallel objects to encapsulate these modules and to distribute them to remote distributed resources. All these three modules will perform the computation in parallel. Passive data access (see chapter 6) is very useful in this case where the data required for each module is not available off line but need to be computed by other modules. The computation is performed in a concurrent method of the parallel object. Inside this method, when a specific data is really needed, it will wait for an event (using the event sub-system, see section 5.2.3 of chapter 5) that will be triggered when the required data from other modules has arrived. When a part of the results is available, it will update other modules (parallel objects) through the ”Set” methods on the target objects.

The Alpine3D coupled by ParoC++ is presented in Fig. 11.6. `AlpineControl` is a sequential object running on the local machine where the user starts the Alpine3D application. `AlpineControl` is used to co-ordinate the computation, the inputs and the outputs of the system over time units. The parallel objects `InputObj`, `OutputObj` and `SnowParameters` are shared objects that are accessible from all computation modules. These objects run on the machine which stores the input data and the output results of the simulation. We assume that this machine is also the local machine where the user starts the application. Three parallel objects `SnowDrift`, `SnowPack` and `EnergyBalance` located on different remote machines contain three concurrent methods ”Compute” that will be asynchronously invoked by `AlpineControl` each time unit. During ”Compute” execution, these objects will also invoke each other to update data when the data is computed.

11.3.2. Second part: parallelization inside modules

To improve considerably the overall performance of Alpine3D, coupling different modules is not enough. We also need to reduce the computation time inside each module. Finer grain parallelism inside each module is required.

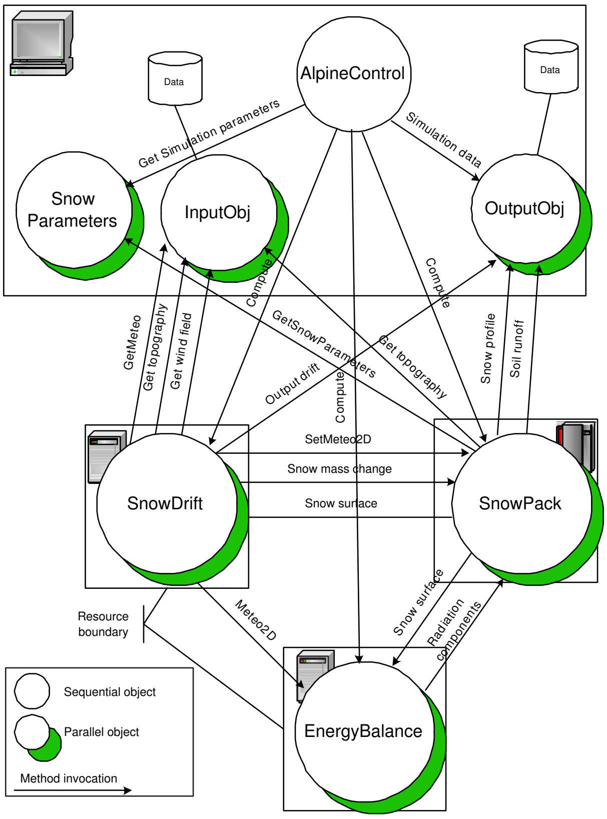


Figure 11.6: Coupling Alpine3D modules using ParoC++

We analyze the computation time of each module. SnowDrift computation is the heaviest part of the Alpine3D. So we have decided to parallelize this module first. At the moment, only SnowDrift is parallelized.

SnowDrift computation consists of three major steps: first, we need to compute the saltation for each grid point (2D grid on the surface); secondly, the saltation results will be used to compute suspension which is a convergence loop over all grid points (3D grid); and finally, the snow mass changes is calculated based on the 3D grid data. The first two steps take most of the computation time. So we will parallelize the calculation of saltation and suspension.

The parallelization is illustrated in Fig. 11.7. We will divide the whole geographical region that we need to simulate the snow process into small regions and we then do the simulation on these small regions in parallel. After each simulation step, information on the bound will be exchanged between neighboring regions. The computation of saltation is rather easy to parallelize since each grid point can be computed independently. The computation of suspension is harder to parallelize because we have to use the information over all 3D grid points to decide the convergence condition. However, this condition can be synthesized from all sub-condition of all sub-regions.

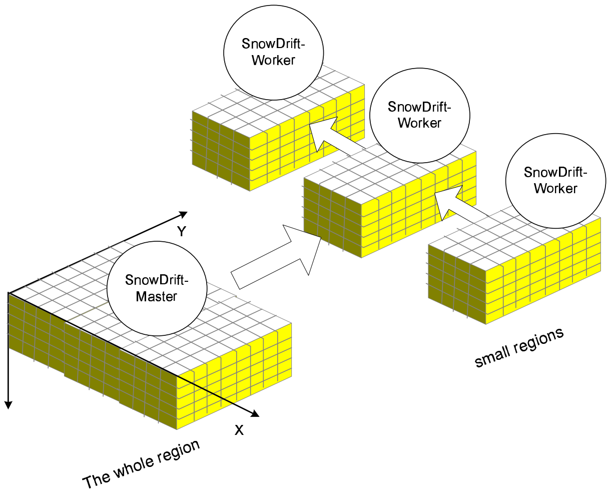


Figure 11.7: Parallelization inside the SnowDrift module

The parallel algorithm is presented in Fig. 11.8. `SnowDriftMaster` contains several `SnowDriftWorker`s which are responsible for computing the saltation and the suspension for sub-regions. `SnowDriftMaster` receives computation requests for the SnowDrift module from the `SimulationControl` in a concurrent method (a thread). It then distributes the computation to workers and waits for the results. The computation consists of three sequent phases: saltation computation, suspension computation and snow mass change computation. All these phases will be executed on the workers. The master will co-ordinate the phases and update the final results.

The ParoC++ event sub-system (see section 5.2.3 of chapter 5) together with the passive data access (see chapter 6) has been used to acknowledge the arrival, and to synthesize the results from workers. It is the worker who will actively update and synthesize the results to the master (via an ”update” method invocation). The ”update” method on the master has a counter (an attribute of the object) that counts the number of updates of workers. When all workers have updated the master, this method will raise an event ”computation complete” so that the ”wait-for-event” on the other concurrent method resumes and the next computation phase will be continued.

11.4. Experiment results

The Hydro@Alpine3D project is ongoing. At the moment, no exhaustive tests has been performed yet. Therefore we only present our preliminary results of the parallel Alpine3D software that simulates snow development in a small area of the Alpine region.

We measure the computation time of the snow development for 120 hours on real data collected from the field (wind data, snow cover data, meteorological data, topography data, etc.). The result is shown in Fig. 11.9. In the figure, the X-axis shows the number of

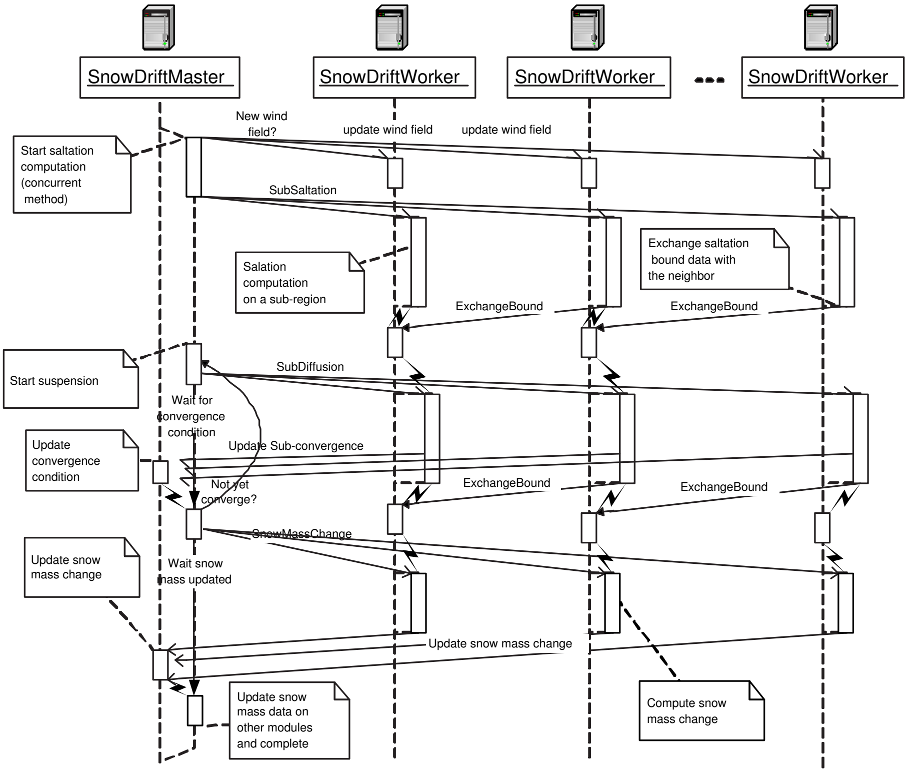


Figure 11.8: UML sequence diagram of the parallel snowdrift computation

processors used for parallel SnowDrift computing; the Y-axis is total simulation time. We test the Alpine3D with parallel SnowDrift computation up to 16 processors.

For the sequential run of Alpine3D on Linux/Pentium 4-2.8GHz, it takes about 90 minutes to complete the simulation. The parallel version of Alpine3D with one processor (no actual parallelization) takes about 98 minutes due to the extra communication cost. With 2 processors, the simulation time is about 55 minutes. The total simulation time continues to decrease as the number of processors increases up to 16. It takes about 14 minutes to finish the whole simulation. However, we notice that increasing number of processors has stronger effect to the overall performance when the number of processors is smaller than or equal 12.

There are three factors that affect the parallel efficiency of the application:

- The time waiting for inputs.
- The data dependencies between modules.
- The overlapping of computation at the bound between two neighbors.

Each wind field in the experiment has about 30MB of data and it is stored locally. The total time spent for reading the inputs (total wind fields: 20) is about 1 minute which is not considerable compared to the total simulation time.

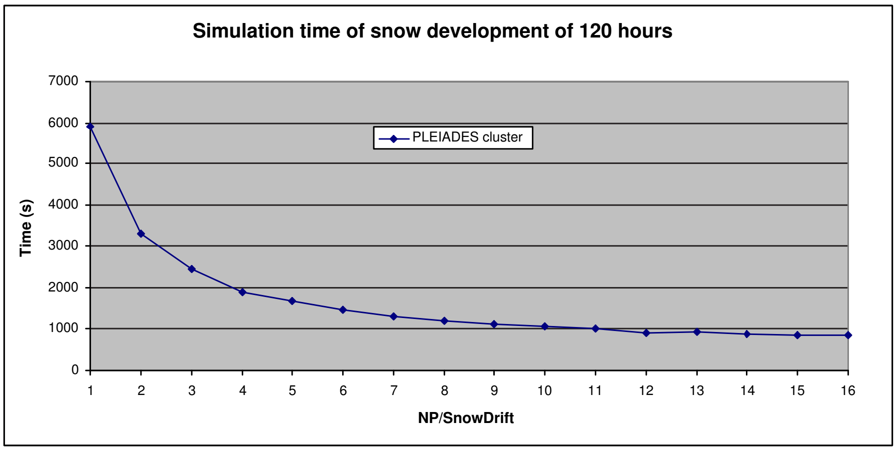


Figure 11.9: Parallel snow development simulation of 120 hours

The data dependencies have a strong influence to the overall performance of the system. As we have mentioned, at the moment, we only focus on parallelizing SnowDrift module-the ”bottle-neck” of the system. When the computation time of SnowDrift decreases, the computation time of other modules becomes considerable. SnowDrift starts to wait for data from other modules when the number of processors used for parallel SnowDrift increases. This problem will be solved when the computation time of other modules will be decreased through the parallelism.

The last factor is the overlap between two neighbor sub-regions. When the number of processors increases, the percentage of overlap over the whole sub-region increases. Also the synchronization (exchange/update results) cost goes up. These extra costs will degrade the overall performance.

This explains why at the current state of the project where only one module has been parallelized, we can speed up the Alpine3D just 6 times (compared to the sequential version) with 16 processors for parallel SnowDrift.

11.5. Summary

The chapter presents the second test case of ParoC++ on the snow simulation and avalanche warning system called Alpine3D developed at the Swiss Federal Institute of Snow and Avalanches Research in Davos.

Alpine3D is not a full functional software yet. Each module is continuously evolved. The programming language heterogeneity is one of the nature of the software where each part is developed by scientists with different backgrounds. How to combine such differences into a single well-integrated application? How to parallelize the application while each component is still in the developing phase? Those questions have been answered by ParoC++.

While in the test case 1, we demonstrated the ability to extract the high performance from the heterogeneity by application adaptation and dynamic reaction to external changes combining with monitoring, ParoC++ in this test case showed an example of how to parallelize, to integrate and to manage a complex existing system. ParoC++ has brought the flexibility, the modularity and the extensibility in an open object-oriented design to the Alpine3D system.

ParoC++ has been used as the principal tool to integrate different heterogeneous modules, to parallelize an existing application and to deploy it on the Grid. Although this is an ongoing project and performance is not the first objective that we want to demonstrate in the test case, primary results of the partial-parallel version of Alpine3D have been presented. These results somehow explain the situation of this complex system and provide us a good start point for continuing the parallelization of other modules to make the overall performance really meaningful for a large scale avalanche warning system for Switzerland.

Chapter 12

Test case 3: Time constraints in Pattern and Defect Detection System

In chapter 10 we presented the implementation of the Pattern and Defect Detection System (PDDS) using ParoC++ without considering the requirement on the time constraints (real-time image analysis). The performance requirement can be achieved by application adaptations. In this chapter, we present another approach based on the ParoC++ framework of the parallelization scheme (chapter 8). This approach is more efficient to deal with highly heterogeneous environments by adapting the size of sub-problems to the heterogeneous capability of resources.

12.1. Algorithms

Figure 12.1 shows the decomposition tree of the parallelization scheme (see chapter 4). The root of the tree is the original image that we need to analyze. The image can be split into three smaller images (image 1, 2 and 3). The image 3 is to detect patterns near the bounder of the two images 1 and 2 that are not able to be detected by both images. Two sub-images 1 and 2 will be further divided until each sub-images is as small as twice the width of the pattern template.

12.2. The parallelization scheme construction

From the decomposition tree in section 12.1, we need to construct three parallel objects: `PatternProb`, a sub-problem of the node inside the decomposition tree, to find patterns on an image; `ImageSource` to acquire images from the external system, split the images into sub-images at different size; and `Output` to output the results to the external system. Each

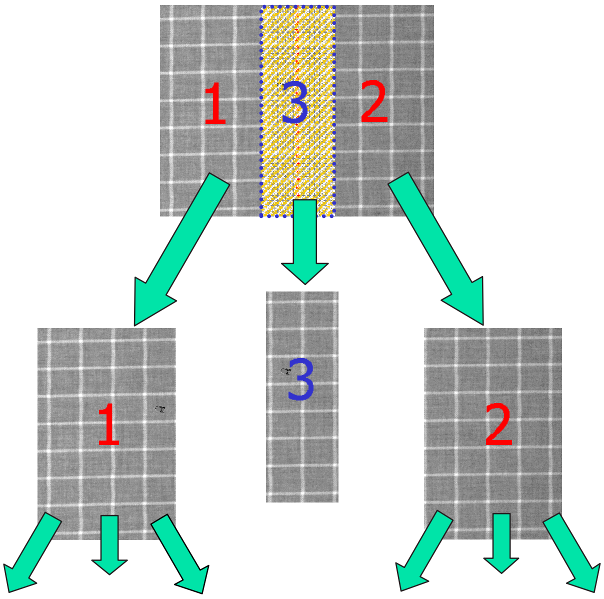


Figure 12.1: Decomposition tree: dividing the image to sub-images

`PatternProb` also stores the ID of the sub-image inside the `ImageSource` that corresponds to the sub-problem at the node of the decomposition tree.

We also need to build the sequential class `PatternNode` which is derived from `DTreeNode`. `PatternNode` is responsible for creating the parallel object `PatternProb` through the virtual method `InitProblem`.

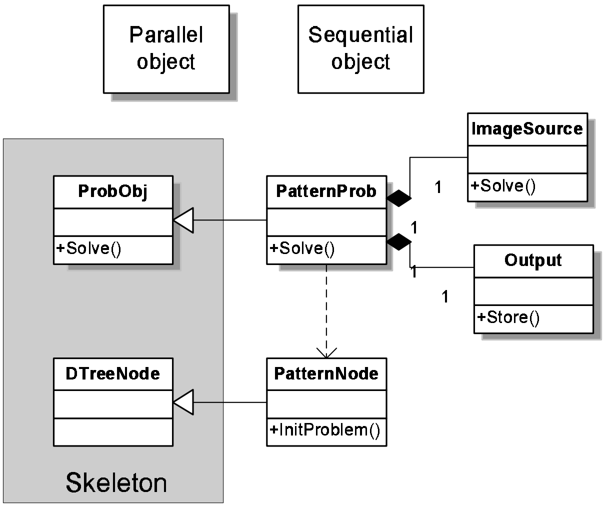


Figure 12.2: The parallel object diagram

Since we use FFT codes to compute the cross-correlation, the complexity of the algorithm is $C.W.H.log(W.H)$ where `C` is a constant value, W and H are the width and the height of the image. The question is how to find the correct value of the constant `C`? In this experiment, we estimate the value of `C` based on the relationship between the benchmark values of ParoC++ service (matrix multiplication) on different platform (Linux, Solaris), the computation time of 2D FFT on different sizes of images on those platforms. From this experiment, we choose C=7.

The construction of the parallelization scheme is illustrated by the following code:

```
/**** Procedure to construct the DT *******/

 PatternNode *ConstructTree(ImageSource &mysrc, Output &myoutput,
     int offset, int imWidth, int imHeight, CImage &mypattern,
     int overlapsize, int minWidth)
{
  float mflops=evaluate the mflops required for the current sub-image;
  PatternNode *node=create new PatternNode with mflops as an argument;

  if (current image is still big enough)
    {
      PatternNode *left=recursive call to ConstructTree on the left image;

      PatternNode *right=recursive call to ConstructTree on the right image;

      node->AddChild(left);
      node->AddChild(right);

      if (the middle image is big enough)
      {
           PatternNode *mid=create PatternNode for the middle image;
          node->AddChild(mid);
        }
    }
  return node;
}
```

`ConstrucTree` will create the current node of the tree with the image ID as a pair *(offset, imWidth)*, where *offset* is the start position of the sub-image within the original image and *imWidth* is the width of the sub-image. It then decides if the decomposition can continue or not. If yes, it will call recursively itself to construct the sub-trees.

To execute the parallelization scheme, as the decomposition tree is constructed, the user only needs to set up the time constraint (method `SetTimeConstraint`), then to instantiate a suitable solution (method Init) and finally to invoke the Solve method on the root node of the tree:

```
tc=input the time constraint from the user;
PatternNode *root=ConstructTree(...);
 root->SetTimeConstraint(tc);

 if (!root->Init())
   {
     printf("Can not instantiate the solution. Fail to solve\n");
     return 1;
   }
 root->Solve();
```

12.3. The results

We run the application on the heterogeneous environment of 130 machines running Linux/Pentium 4 and Solaris/Sparc with two assumptions:

- The user only knows one ParoC++ access point to access the ParoC++ environment. He does not have any knowledge about the machine inside the environment.
- The initial topology of machines is the same as the one we described in section 9.4 of chapter 9.

Before doing this experiment, we have evaluated the performance of each machine by a matrix multiplication benchmark of size 500x500. This benchmark result will be used by ParoC++ services as the computing power of the resource to match with the resource requirements from the PatternProb objects at run time. Values reported by the benchmark show that inside the test environment, Linux/Pentium 4 systems run about twice as fast as the Solaris/sparc systems.

The input data set consists of a chain of 100 image frames (8-bit gray images), each has the size of 2048x2048. We need to analyze all these images to find pattern positions.

In each run, the user will specify the time constraint within which all 100 images should be analyzed. The parallelization scheme framework will be applied to solve the problem and the actual computation time is measured and compared to the time constraint. We also record the number of decomposed problems which somehow reflects the heterogeneous grain of parallelism of the run.

The results are shown in Fig. 12.3. The experiment consists of 16 runs with different time constraints for all 100 images to be computed ranging from 600 seconds down to 30 seconds. The X-axis corresponds to the run number. We start the first test with the time constraint

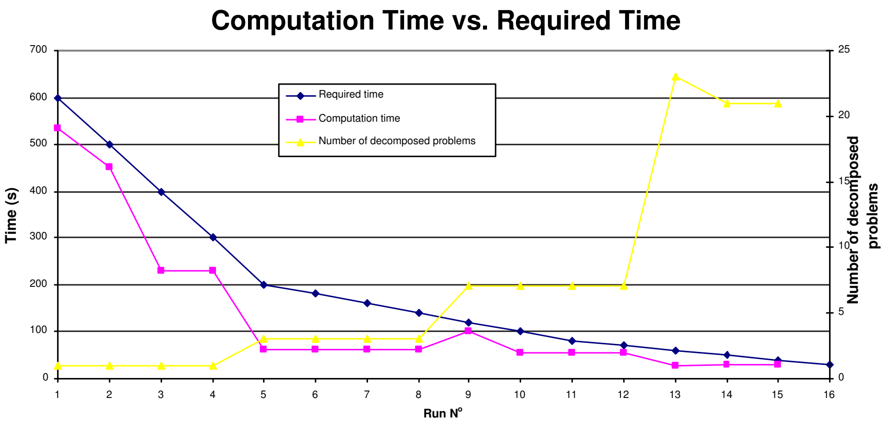


Figure 12.3: The time constraint vs. the actual computation time

of 600 seconds (6 seconds/image). ParoC++ system finds a resource (a Solaris machine) with enough computing power to analyze the whole image chain sequentially (actual: 536sec, constraint: 600sec). The second test requires the time constraint of 500 seconds and the actual computation time is 453 seconds on 1 machine (another Solaris machine). For the runs #1 to #4, ParoC++ system finds some resource with enough computing power to solve the problem sequentially.

From run #5 (time constraint: 200 seconds), the problem starts to be decomposed. Run #5 to run #8 decompose each image into 3 sub-images (first level decomposition: 2 half images and the overlap). The computation time in all 4 runs is about 62 seconds (vs. time constraints from 200 down to 140 seconds). The image is decomposed into more sub-images as the time constraint is decreased: run #9 to run #12: 7 sub-images; run #13: 23 sub-images and run #14-15: 21 sub-images. At run #16, the ParoC++ framework reported that it can not instantiate a solution due to the lack of resources.

Two points that we can figure out easily from the results: firstly, at run #9, the actual computation time is about 100 seconds (time constraint: 120 seconds) which is bigger than that of run #8 (61 seconds). As we have clearly stated in section 4.3.1 of chapter 4, we do not aim at finding the fastest solution but we will find a solution that satisfies the time constraint. Selecting an acceptable solution (satisfy the time constraint, not the best) is the case of run #9. The solution depends on the distributed resource discovery algorithm of the ParoC++ system (see section 7.6). Secondly, at run #13, the number of sub-problems is greater than those of runs #14 and #15. This is because we tried to simulate the dynamics of the run-time environment by restarting a Linux machine just before run #13 started. This event leads the parallelization scheme to find slower Solaris machines and hence it needs to decompose further in order for the time constraint to be satisfied.

Although there are some differences in performance between different runs but the time constraints in the experiment have been satisfied. The experiment has illustrated how to use the ParoC++ parallelization scheme for solving problems with time constraints.

12.4. Summary

The chapter completes the discussion about the parallelization scheme for solving time constrained problems on the Grid by a test case from industrial image processing. We described how to elaborate the parallelization scheme using the ParoC++ framework for time constrained problems. The experiment on the mixed Solaris/Linux environment illustrates the ability to automatically and transparently derive different grains of parallelism based on the time constraint from the user and the available resources inside the execution environment.

Chapter 13

Conclusion

The dissertation presents a new parallel programming paradigm for developing high performance (HPC) applications on the Grid. We address the question "How to tailor the HPC applications to the Grid?" where the heterogeneity and the large scale of resources are the two main issues. We answer the question at two levels: the programming tool level and the parallelization concept level.

At the programming tool level, the state-of-the-art of Grid computing shows that currently, there is no suitable programming tool to extract the performance from the Grid. We address this issue through the application adaptation in the *parallel object model* and the *ParoC++ programming system* under two forms: either the application components should somehow decompose dynamically based on the available resources of the environment; or the components should be able to ask the infrastructure to select automatically the suitable resources by providing some descriptive information about the resource requirements.

The *parallel object model* is the generalization of the sequential object model with the integration of user requirements via *object-description* into the *shareable distributed object.* We refer such objects as *parallel objects.* Two important properties of the parallel objects are: dynamic parallel object creation and destruction; and requirement-driven object allocation. The first property enables the application to react to the changes inside the environment or from the user by acquiring or releasing additional resources for the application. The second property provides a mean for application to describe the performance requirement it needs from the executing environment.

We have implemented the parallel object model in the ParoC++ programming system. ParoC++ provides a comprehensive object-oriented infrastructure for developing applications, for managing the Grid environment and for executing the application on the Grid. ParoC++ consists of a programming language extended from C++ to support parallel objects, a compiler to compile the ParoC++ source code and the run-time system to run ParoC++ applications on the Grid. We focus on the open design and extensibility of the system so that ParoC++ can be used to glue different Grid toolkits. The integration of

Globus into ParoC++ has been used as an example of this open and extensible design.

At the parallelization concept level, we investigate the *parallelization scheme* which provides the user a method to express the parallelism for a class of time constraint problems with a known (or well-estimated) complexity on the Grid. The parallelization scheme is constructed on the two principal elements: the *decomposition tree* and the *decomposition dependency graph*. The decomposition tree defines multi-level problem decompositions that represent all feasible solutions to the original problem. The decomposition dependency graph shows the partial order of execution of sub-problems. Algorithms on the parallelization scheme have been developed to compute the time constraints, the resource requirements and to instantiate automatically and transparently a suitable solution based on the available resources of the environment.

The ParoC++ framework for solving time constraint problems is result of the integration of ParoC++ and the parallelization scheme. The framework provides a high-level abstraction based on parallel objects for the user to express the parallelism and the time constraint, to implement the time constrained application and to execute the application on the Grid. With the framework, users can focus on the ”logic” of the problems and leave all of the complexities of the Grid environment over the framework and the ParoC++ system.

The thesis emphasizes on the high performance computing issue on the Grid. Many other challenges of the Grid are just partially covered or not covered: security, resource accounting, efficiency evaluation of the Grid system and the Grid application, benchmarking, application fault tolerance, etc. Although they are important issues of the Grid.

We assert the dissertation with a series of benchmarks and with two real life applications on image processing and on snow simulation and avalanche warning. The results show the effectiveness of ParoC++ on developing high performance computing applications and in particular for solving the time constrained problems on the Grid.

The thesis opens some new directions for the future of high performance computing on the Grid: from the parallel object model, is that possible to automatically parallelize an object-oriented application? from the ParoC++ system, how to evaluate the efficiency of the system? how to verify the properness, to find deadlocks and to detect performance bottleneck inside the application? from the parallelization scheme, what is the resource requirement metric? how to evaluate the complexity of time constrained applications? And above all, we would like to make ParoC++ as the pioneer in bringing object-oriented methods and high performance computing all together on the Grid environment.

Appendix A

Genetic algorithm for the Min-Max problem

We find an approximate solution of $T_1^s, T_2^s, \ldots, T_m^s$ to (4.7) with the conditions (4.2) and (4.4) presented in chapter 4.

A.1. The Algorithm

The algorithm is described as follow: the population consists of W individuals. Each individual is visualized as a circle with circumference of αT. The circle is split into m sectors whose lengths are $T_1^s, T_2^s, \ldots, T_m^s$. By this representation, the constraints (4.2) and (4.4) are satisfied. Initially, all W individuals are randomly selected. The evolution process is performed by mutation and crossover operations on the population with the corresponding probabilities ρ_1 and ρ_2.

Mutation. For an individual D, we randomly select a sector T_i^s of the circle and increase by x% (x is a random number in range $< -100 \ldots 100 > \setminus \{0\}$, negative value of x means ”decrease T_i^s”). All other sectors $T_j^s (j \neq i)$ will be adjusted accordingly:

$$T_j^{\prime s} = \begin{cases} T_j^s \left(1 + \frac{x}{100}\right), & \text{if i=j;} \\ T_j^s \left(1 - \frac{x T_i^s}{100(\alpha T - T_i^s)}\right), & \text{otherwise.} \end{cases}$$

Crossover. This operation consists of 2 steps: first, randomly select two individuals from the population, select the cut index on the circle and swap two parts of the two individuals to generate the new generation (see fig. A.2); then normalize the other parts of the circle by shrinking or expanding so that the new circles have the same circumference αT.

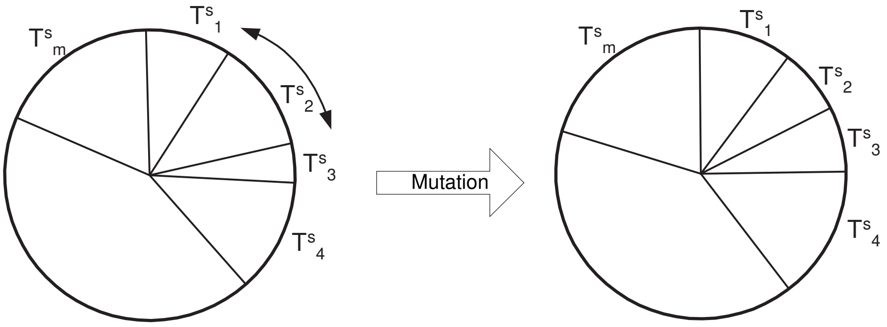


Figure A.1: Mutation operation

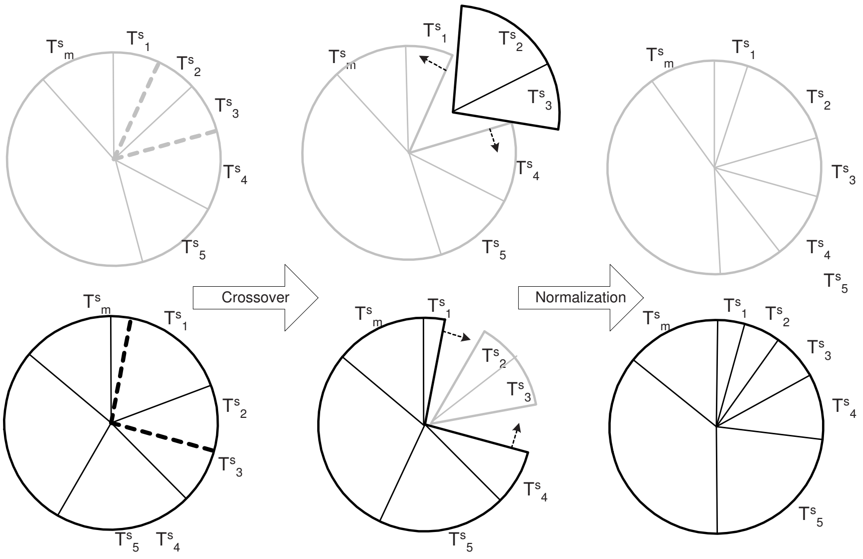


Figure A.2: Crossover operation between two individuals

After performing mutation and crossover, a new generation is created. For each individual, a fitness function obtained from (4.7) will be evaluated:

$$F(T_1^s, T_2^s, \dots, T_m^s) = max\{g_1(\mathcal{C}(P_1), T_1), g_2(\mathcal{C}(P_2), T_2), \dots, g_n(\mathcal{C}(P_n), T_n)\} \tag{A.1}$$

Fitness function shows how "good" an individual is: the smaller the fitness value, the better the match of the individual to the target. In order to keep the fix size population after performing crossover and mutation, we remove "bad" individuals with the biggest values of fitness.

The evolution process stops when a number of generations has been reached or the "best" individual does not improve after a number of iterations (e.g. 100 iterations).

Table A.1: Genetic Algorithm on Simple Data Set

Size:	200	400	600	800	1000	1200	1400
Number of epoch:	23199	35396	46798	52197	58899	60200	81297
MaxPower(GA):	964.0	2040.6	2958.9	4054.7	5044.8	6032.4	7135.2
MaxPower(Optimal):	963.9	2040.1	2957.8	4052.4	5041.6	6026.4	7130.2

Table A.2: Genetic Algorithm on Complex Data Set

Sub problems:	200	400	800	1200	1600	2000	2400	2800
Number of epoch:	2698	3896	8100	5290	8197	12500	13097	14997
Computation time:	0m07s	0m28s	2m59s	4m02s	10m33s	24m09s	36m17s	56m01s

A.2. Experimental results

We have performed the genetic algorithm with the following parameters:

- Population size: $W = 200$
- Mutation probability: $\rho_1 = 0.4$
- Crossover probability: $\rho_2 = 0.2$
- Stop criteria: after 100000 generations or when the best individual does not improve after 100 iterations.

The input data is a sequential diagram randomly generated. We follow the two experiments: first we generate a simple data set in which each problem P_i spans exactly one step. In this case, the optimal solution to (4.7) can be calculated using (4.8). The results are then compared with the results obtained by using the genetic algorithm. The second experiment deals with the performance of the algorithm on more complex data set where the number of steps is smaller than the number of sub-problems. In both cases, the complexities of sub-problems are randomly generated, the time constraint is $T = 1$ and the constraint guard coefficient is $\alpha = 1$. For the latter case, the sequential diagram is also randomly generated such that the number of sub-problems is twice the number of steps (average of two sub-problems to be solved in each step).

The result for the simple data set is shown in Table A.1 where the number of sub-problems is also the number of steps. $MaxPower$ is the return value of function F in (A.1). The genetic algorithm gives good results in compared to the optimal solution. In all cases, the difference is not considerable (about 0.1% bigger).

Table A.2 shows the convergent speed of the genetic algorithm in the second experiment. All tests are done on a Linux/Pentium 4, 1.7GHz machine. The convergence speed depends not only on the number of sub-problems but also on the connectivity between sub-problems in the sequential diagram. It is quite fast when the number of sub-problems in the sequential diagram is small (7 seconds for 200 sub-problems/the min-max problem of 100 steps or variables). This increases up to about 56 minutes for a decomposition of 2800 sub-problems (the min-max problem with 1400 variables).

Bibliography

[1] Bill Allcock, Joe Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, 2001. [55]

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. *GridFTP Protocol Specification.* GGF GridFTP Working Group Document, September 2002. http://www.globus.org/research/papers.htm. [55]

[3] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992. [17]

[4] H. Balen. *Distributed object architectures with CORBA.* Cambridge University Press, 2000. [16]

[5] David Barkai. *Peer-to-Peer Computing: Technologies for Sharing and collaborating on the Net.* Intel Press, 2002. [34]

[6] Perry Bartelt and Michael Lehning. A physical SNOWPACK model for the Swiss avalanche warning-Part I: numerical model. *Cold Regions Science and Technology*, 35:123–145, 2002. [110]

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, (1):173–189, 1972. [33]

[8] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990. [56]

[9] T. D. Braun, H. J. Siegel, and A. A. Maciejewski. Mapping heuristics for tasks with dependencies, priorities, deadlines and multiple versions in heterogeneous environments. In *Proc. of the 16th International Parallel and distributed Processing Symposium*, 2002. [29]

[10] J.-P. Cagnard. The parallel cellular programming model. In *The 8^{th} euromicro workshop on Parallel and Distributed Processing*, 2000. [15]

[11] J.-P. Cagnard. *ParCel-2 : un modèle de programmation parallèle, cellulaire, hiérarchique.* PhD thesis, Swiss Federal Institute of Technogoly-Lausanne, 2001. [15]

[12] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000. [16]

[13] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991. [14]

[14] John B. Carter, Dilip Khandekar, and Linus Kamb. Distributed shared memory: Where we are and where we should be headed. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 119–122, 1995. [14]

[15] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997. [69]

[16] Ethan Cerami. *Web services essentials.* O'Reilly Press, 2002. [10]

[17] CERN. *The Large Hadron Collider Project.* http://lhc.web.cern.ch/lhc/. [8, 55]

[18] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, (23):187–200, 2001. [55]

[19] R.S. Chin and S.T. Chanson. Distributed object-based programming system. *ACM Computing Surveys*, 23(1), 1991. [16]

[20] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000. [69]

[21] A. Corradi, L. Leonardi, and F. Zambonelli. Hpo: a programming environment for object-oriented metacomputing. In *Proc. of the 23rd EUROMICRO conference*, 1997. [17]

[22] A. Corradi, L. Leonardi, and F. Zambonelli. Parallel object allocation via user-specified directives: A case study in traffic simulation. *J. Parallel Computing*, (27):223–241, 2001. [17]

[23] Caroline Cruz-Neira, Daniel J. Sandin, and Tom DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In *Proc. of SIGGRAPH 93*, August 1993. [59]

[24] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998. [75, 80]

[25] Gilles Fedak, Cécile Germain, Vincent Néri, and Franck Cappello. Xtremweb : A generic global computing system. In *CCGRID2001, workshop on Global Computing on Personal Devices*, 2001. [61]

[26] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, pages 365–375, 1997. [69]

[27] I. Foster and N. Karonis. A grid-enabled mpi: Message passing in heterogeneous distributed computing systems. In *Proc. 1998 SC Conference*, November 1998. [2, 13]

[28] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997. [1, 9, 13, 61, 80]

[29] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998. [1, 7, 8, 11, 61]

[30] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002. [10, 11, 80]

[31] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3), 2001. [1, 34]

[32] I. Foster, A. Roy, and V. Sander. A quality of service architecture that combines resource reservation and application adaptation. In *The 8th International Workshop on Quality of Service*, 2000. [25]

[33] B. Françoise, C. Denis, F. Nathalie, and S. David. Optimizing remote method invocation with communication-computation overlap. *Future Generation Computer Systems*, 18:769–778, 2002. [56]

[34] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine-A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. [13, 56]

[35] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998. [14]

[36] Chris Gill, Fred Kuhns, Douglas C. Schmidt, and Ron Cytron. Empirical differences between cots middleware scheduling paradigms. In *The 8th IEEE Real-Time Technology and Applications Symposium*, September 2002. [25]

[37] Globus Project. *Globus GRAM Documentation.* http://www-unix.globus.org/developer/resource-management.html. [80]

[38] A. S. Grimshaw and W. A. Wulf. Legion — a view from 50,000 feet. In *Proc. of the 5th IEEE International Symposium on High Performance Distributed Computation*, August 1996. [1, 9]

[39] A. S. Grimshaw, W. A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997. [11]

[40] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: An operating system for wide-area computing. *IEEE Computer*, 32:5:29–37, May 1999. [11, 13]

[41] Andrew Grimshaw, Adam Ferrari, and Emily West. *Parallel Programming Using C++*, pages 383–427. The MIT Press, Cambridge, Massachusetts, 1996. [2, 17]

[42] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, sep 1996. [13, 95]

[43] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6. [95]

[44] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. Analysis of a class of parallel matrix multiplication algorithms. In *Proc. of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 110–116, 1998. [29]

[45] R. Henderson and D. Tweten. *Portable Batch System: External reference specification.* Ames Research Center, 1996. [75]

[46] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998. [15]

[47] G. Hoo, W. Johnston, I. Foster, and A. Roy. Qos as middleware: Bandwidth reservation system design. In *Proc. of the 8th IEEE Symposium on High Performance Distributed Computing*, 1999. [25]

[48] P. Jędrzejowicz and I. Wierzbowska. Scheduling multiple variant programs under hard real-time constraints. *European Journal of Operational Research*, 127:458–465, 2000. [29]

[49] Elizabeth Johnson and Dennis Gannon. HPC++: Experiments with the parallel standard template library. In *International Conference on Supercomputing*, pages 124–131, 1997. [18]

[50] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003. [13]

[51] K. Keahey and D. Gannon. PARDIS: A parallel approach to CORBA. In *The 6th IEEE International Symposium on High Performance Distributed Computing*, August 1997. [18]

[52] P. Kuonen, G. Babin, N. Abdennadler, and P-J. Cagnard. Intensional high performance computing. *Lecture Notes in Computer Science*, 1830, 2000. ISBN 3-540-67647-3. [15]

[53] P. Kuonen, T. A. Nguyen, and J.-Ph. Thiran. Le projet ForAll ou l'analyse d'images au service de la mode. *EPFL Flash Informatique*, 6, 2003. [104]

[54] J. Lee, B. Tierney, and W. Johnston. Data intensive distributed computing: A medical application example. *Lecture Notes in Computer Science*, 1593, 1999. [55]

[55] Michael Lehning, Perry Bartel, Bob Brown, and Charles Fierz. A physical SNOWPACK model for the Swiss avalanche warning- Part III: meteorological forcing, thin layer formation and evaluation. *Cold Regions Science and Technology*, 35:169–184, 2002. [110]

[56] Michael Lehning, Perry Bartelt, Bob Brown, Charles Fierz, and Pramod Satyawali. A physical SNOWPACK model for the Swiss avalanche warning-Part II: Snow microstructure. *Cold Regions Science and Technology*, 35:147–167, 2002. [110]

[57] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and distributed Computing*, 59(2):107–131, November 1999. [29]

[58] Microsoft Corporation. *Distributed Component Object Model.* http://www.microsoft.com/com/tech/dcom.asp. [16]

[59] Nuno Neves, Miguel Castro, and Paulo Guedes. A checkpoint protocol for an entry consistent shared memory system. In *Symposium on Principles of Distributed Computing*, pages 121–129, 1994. [14]

[60] T. A. Nguyen, P. Kuonen, and J.-Ph. Thiran. *FORALLWEAR Project: Deliverable D.3.5: Report on defect detection algorithms and performance analysis.* [104]

[61] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, 1991. [14]

[62] Object Management Group. *Real-Time CORBA specification.* http://www.omg.org. [29]

[63] Guedes Paulo and Castro Miguel. Distributed Shared Object Memory. In *Proc. 4th Wshop. on Workstation Operating Systems (WWOS-IV)*, Napa, CA (USA), 1993. IEEE Computer Society Press. [14]

[64] C. Petitpierre. Synchronous C++, a language for interactive applications. *IEEE Computer*, pages 65–72, September 1998. [17]

[65] T. Priol and C. Rene. COBRA: A CORBA-compliant programming environment for high-performance computing. In *Proc. of Europar'98*, pages 1114–1122, Southampton, UK, September 1998. [18]

[66] Raman R., Livny M., and Solomon M. Matchmaking: Distributed resource management for high throughput computing. In *The 7th IEEE International Symposium on High Performance Distributed Computing*, 1998. [69]

[67] S. K. Reinhard, R. W. Ple, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proc. of the 23rd Annual Symposium on Computer Architecture*, May 1996. [56]

[68] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, Jan.-Feb. 2002. [69]

[69] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen. MPICH-GQ: Quality-of-service for message passing programs. In *Proc. of the IEEE/ACM SC2000 Conference*, November 2000. [2, 13]

[70] Ben Segal. Grid computing: The european data grid project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, pages 15–20, October 2000. [55]

[71] M. Shirts and V.S. Pande. Screensavers of the world, unite! *Science*, 2000. [8]

[72] M. Snir and W. Gropp et al. *MPI: The Complete Reference.* MIT Press, 1998. [56]

[73] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman, and B. Tierney. File and object replication in data grids. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC2001)*, 2001. San Francisco, California. [55]

[74] Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001. [69]

[75] Sun Microsystems. *Remote Method Invocation specification.* ftp://ftp.javasoft.com/docs/jdk1.2/security-spec.pdf. [16]

[76] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *ICDCS'03*, 2003. [14]

[77] David A. Thurman. *JavaPVM: The Java to PVM Interface. HTML Document*, June 1996. http://homer.isye.gatech.edu/chmsr/JavaPVM/. [16]

[78] B. Tierney, W. Johnston, and J. Lee. A cache-based data intensive distributed computing architecture for grid applications. In *CERN School of Computing*, September 2000. [55]

[79] Weiqin Tong, Jingbo Ding, and Lizhi Cai. A parallel programming environment on grid. In *International Conference on Computational Science 2003*, pages 225–234, 2003. [16]

[80] S. Tuecke, D. Engert, I. Foster, M. Thompson, L. Pearlman, and C. Kesselman. *Internet X.509 Public Key Infrastructure Proxy Certificate Profile.* IETF, 2001. [80]

[81] UC Berkeley. *SETI@home project.* http://setiathome.ssl.berkeley.edu/index.html. [8]

[82] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. [15]

[83] Vijay Pande, et al. Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. *Peter Kollman Memorial Issue, Biopolymers*, 2002. [8]

[84] W3C. *Simple Object Access Protocol (SOAP) 1.1.* http://www.w3.org/TR/SOAP/. [10]

[85] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In IEEE Press, editor, *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, 2003. [80]

[86] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. *Lecture Notes in Computer Science*, 1800, 2000. [16]

[87] Bojan Zagrovic, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. Simulation of folding of a small alpha-helical protein in atomistic detail using worldwidedistributed computing. *Journal of Molecular Biology*, 2002. [8]

[88] S. Zhou. Lsf: Load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing*, 1992. [75]

Curriculum Vitae

Tuan-Anh Nguyen
Computer Science Theory Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
1015 Lausanne, Switzerland
Email: `tuananh.nguyen@epfl.ch`
Web: `http://lithwww.epfl.ch/~tanguyen`

Education

2001 - 2005 Ph.D. candidate at School of Computer and Communication Sciences, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland.

1993 - 1998 Bachelor of Engineering in Computer Science, University of Technology, Ho Chi Minh city, Vietnam.

Professional Experience

1998 - 1999 **Research and Teaching Assistant**, Department of Information Technology, University of Technology, Ho Chi Minh city, Vietnam

1999 - 2001 **Internship study**, Computer Science Theory Laboratory, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland

2001 - 2004 **Research assistant**, University of Applied Sciences Valais (HEVs), Switzerland

2004 - 2005 **Research assistant**, University of Applied Sciences of Fribourg (EIA-FR), Switzerland

Awards and Honors

- First rank honor for the entrance examination of university by the Chairman of Ho Chi Minh City People's Committee (1993).

- Best students awards by the Rector of University of Technology (1995-1997)
- Two-year scholarship award from Swiss Government (1999-2001)

Personal

Date of birth:	November 27, 1975.
Nationality:	Vietnam.
Civil status:	Single.

Publications

1. T. A. Nguyen, P. Kuonen. *ParoC++: A Requirement-driven Parallel Object-oriented Programming Language.* Future Generation Computer Systems, Elsevier, to appear.

2. T. A. Nguyen, P. Kuonen. *Parallelization Scheme for an Approximate Solution to Time Constraint Problems.* The International Conference on Computational Science 2003 (ICCS2003), 2003, St. Petersburg, Russia.

3. T. A. Nguyen, P. Kuonen. *ParoC++: A Requirement-driven Parallel Object-oriented Programming Language.* The 8th International Workshop on High-Level Parallel Programming Models and Supportive Environments/IPDPS, 2003, Nice, France.

4. T. A. Nguyen, P. Kuonen. *An Object-Oriented Framework for Efficient Data Access in Data Intensive Computing.* The 5th workshop on Advances in Parallel and Distributed Computational Models, 2003, Nice, France.

5. T. A. Nguyen, P. Kuonen. *A Model of Dynamic Parallel Objects for Metacomputing.* The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, Las Vegas, Nevada, USA.

6. T. A. Nguyen. *Distributed Access to Swiss-Tx Using Globus.* EPFL/LITH Report No 126, March 2000

7. T. A. Nguyen. *A Method of Parallel Computing Permeability Field on Distributed Network.* International workshop "Some Problems in Scientific Computing", 1998, Hanoi, Vietnam.