ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

CATEGORIZATION AND VISUALIZATION OF PARALLEL PROGRAMMING SYSTEMS

M.Sc. Thesis by

Ayşe Beliz ŞENYÜZ

Department: Computer Engineering

Programme: Computer Networks

JANUARY 2005

ISTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

CATEGORIZATION AND VISUALIZATION OF PARALLEL PROGRAMMING SYSTEMS

M.Sc. Thesis by

Ayşe Beliz ŞENYÜZ

(504021529)

Date of submission: 27 December 2004

Date of defence examination: 26 January 2005

Supervisor (Chairman): Prof. Dr. A. Emre HARMANCI

Members of the Examining Committee: Prof. Dr. Nadia ERDOĞAN

Assoc. Prof. Dr. Hasan DAĞ

JANUARY 2005

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

PARALEL PROGRAMLAMA SİSTEMLERİNİN SINIFLANDIRILMASI VE GRAFİK GÖSTERİMİ

Yüksek Lisans Tezi

Ayşe Beliz ŞENYÜZ

(504021529)

Tezin Enstitüye Verildiği Tarih: 27 Aralık 2004

Tezin Savunulduğu Tarih: 26 Ocak 2005

Danışmanı: Prof. Dr. A. Emre HARMANCI

Diğer Jüri Üyeleri: Prof. Dr. Nadia ERDOĞAN

Doç. Dr. Hasan DAĞ

OCAK 2005

PREFACE

First of all, I would like to thank Prof. Dr. Emre Harmancı who accepted to be my supervisor. Then, I would like to thank Prof. Dr. Claudia Leopold for giving me the opportunity to work in her group of "Programming Languages and Parallel Programming" at University of Kassel and never let me alone even in the rest of my work in Turkey. Working at University of Kassel was a great experience for me. I feel fortunate to have friends who were always helpful to me in my first time in Germany and who made my time enjoyable, Michael Süss, Björn Knafla. Thanks to Prof. Dr. Nadia Erdoğan who accepted to work with me when I turned back to Turkey.

I am grateful to Dr. Turgay Altılar who encouraged me for the presentations and helped me a lot for the interpretation of my graphics. I am also grateful to Dr. Onur Tan who let me rid off spending a lot of time for formatting the thesis and my colleague Haris Saybaşılı who is always a helpful friend.

Special thanks to my parents Osman Faruk Şenyüz and Şükran Şenyüz who always supported me.

December 2004 Ayşe Beliz Şenyüz

TABLE OF CONTENTS

ABBREVIATIONS viii
LIST OF TABLES ix
LIST OF FIGURES ix
ÖZET x
ABSTRACT x
1. Introduction 1
1.1. Motivation 1
1.2. Contributions 3
1.3. Outline 3
1.4. Disclaimer 3
2. Parallel Programming and Existing Classifications 4
2.1. Parallel Programming 4
2.1.1. Motivation for Parallel Programming 4
2.1.2. Designing Parallel Programs 4
2.2. Existing Classifications 6
2.2.1. Classification by Abstraction Level and Communication Model 6
2.2.1.1. Data-Parallel Systems 6
2.2.1.2. Shared memory Systems 6
2.2.1.3. Message Passing Systems 7
2.2.1.4. Coordination Systems 7
2.2.1.5. Object-Oriented Systems 7
2.2.1.6. High-Level Programming Systems 7
2.2.1.7. Logic programming 8
2.2.2. Classification of Advanced Environments 8
2.2.2.1. Programming Environments 9
3. New Research on Parallel Systems 10
3.1. Research Directions in Parallel Programming 10
3.1.1. Code Correction and Debugging Tools for Systems Using Shared Memory Communication 10

3.1.2. Algorithm Design for Systems Using Shared Memory Communication 11
3.1.3. Mobile Agents 11
3.1.4. Design Patterns and Skeletons 11
3.1.5. Layered Systems 12
3.2. Overview of the Parallel Programming Systems 12

4. Classification of Parallel Systems 21
4.1. Classification according to the Implementation Type 21
4.1.1. Libraries 21
4.1.2. Languages 21
4.1.3. Compiler Directives 22
4.1.4. Parallelizing Compilers 23
4.2. Classification according to the Programming Languages 23
4.2.1. Parallel Programming with Imperative Languages 23
4.2.2. Functional Parallel Programming 24
4.2.3. Object-oriented Parallel Programming 25
4.2.4. Parallel Programming with Logic Languages 26

5. Wiki Engine 27
5.1. The structure of the tool 27
5.1.1. Syntax 27
5.1.2. Use of the Syntax 28
5.1.2.1. Sub-categorization 32
5.2. Systems 32
5.2.1. Imperative Parallel Programming Systems 32
5.2.2. Functional Parallel Programming Systems 32
5.2.3. Object-Oriented Parallel Programming Systems 33
5.2.4. Logic Parallel Programming Systems 33

6. Comparison of MPI and OpenMP 34
6.1. Algorithms 34
6.1.1. Matrix-Vector Multiply 34
6.1.2. Mergesort 35
6.2. Parallel Machines 35
6.2.1. SMP 35
6.2.2. ccNUMA 36
6.2.3. Beowulf Cluster 37
6.3. Performance Comparison 38
6.3.1. Matrix-Vector Multiply Algorithm 39
6.3.2. Mergesort Algorithm 40
6.3.3. OpenMP vs MPI 43

7. Conclusion **45**

7.1. Summary **45**

7.2. Outlook **45**

REFERENCES **47**

APPENDIX **52**

A. Time Measurement for Matrix - Vector Multiply **53**

A.1. OpenMP, SMP **53**

A.2. OpenMP, ccNUMA **55**

A.3. MPI, ccNUMA **58**

A.4. MPI, Beowulf **59**

B. Time Measurement for Mergesort **60**

B.1. OpenMP, ccNUMA **60**

B.1.1. Best Case 60
B.1.2. Random Case 62
B.1.3. Worst Case 64

B.2. MPI, ccNUMA **66**

B.2.1. Best Case 66
B.2.2. Random Case 68
B.2.3. Worst Case 71

B.3. MPI, Beowulf, Small N **74**

B.3.1. Best Case 74
B.3.2. Random Case 74
B.3.3. Worst Case 75

B.4. MPI, Beowulf, Large N **75**

B.4.1. Best Case 75
B.4.2. Random Case 75
B.4.3. Worst Case 76

BIOGRAPHY **77**

ABBREVIATIONS

API	:	Application Programming Interface
MIMD	:	Multiple Instruction Multiple Data
SIMD	:	Single Instruction Multiple Data
SISD	:	Single Instruction Single Data
MPMD	:	Multiple Program Multiple Data
SPMD	:	Single Program Multiple Data
SPSD	:	Single Program Single Data
MPL	:	MasPar Language
HPF	:	High Performance Fortran
NESL	:	Nested Language
MPL	:	MasPar Language
POOMA	:	Parallel Object-Oriented Methods and Applications
UPC	:	Unified Parallel C
pSather	:	Parallel Sather
WPP	:	Whole Program Paths
EREW PRAM	:	Explicit Read Explicit Write Parallel Random Access Machine
MPI	:	Message Passing Interface
TPO	:	Tübingen Parallel Objects
ICC	:	Intel C Compiler
PUFF	:	Parallelization Using Farmed Functions
PMLS	:	Parallel Standard ML with Skeletons
Scampi	:	Simple Caml Interface to MPI
CML	:	Concurrent ML
GpH	:	Glasgow Parallel Haskel
HDC	:	High Order Divide And Conquer Language
P3L	:	Pisa Parallel Programming Language
ALWAN	:	A Language With A Name
eSkel	:	Edinburgh Skeleton Library
CML	:	Concurrent ML
Id	:	Irvine Dataflow
Sisal	:	Streams and Iterations in a Single Assignment Language
pH	:	Parallel Haskell
PCN	:	Program Composition Notation
pH	:	Parallel Haskell
TwoL	:	Two Level
SCN	:	Structured Coordination Language
COPS	:	Correct Object-Oriented Pattern-Based Programming System
CO_2P_3S	:	Correct Object-Oriented Pattern-Based Programming System
SMP	:	Symmetric Multiprocessors
ccNUMA	:	cache coherent Non Uniform Memory Access
PE	:	Programming Environment
PSE	:	Problem Solving Environment

LIST OF TABLES

Page No:

Table 4.1 Imperative Parallel Programming Systems 24
Table 4.2 Object-Oriented Parallel Programming Systems 26
Table 4.3 Logic Parallel Programming Systems 26

LIST OF FIGURES

Page No:

Figure 4.1 Tree of Functional Parallel Programming Systems 25

Figure 5.1 Wiki Main Page 29
Figure 5.2 Editable wiki page 30
Figure 5.3 New Link, New System 30
Figure 5.4 Tree Syntax 31
Figure 5.5 New Categorization Tree 31
Figure 5.6 Tree of Imperative Parallel Programming Systems 32
Figure 5.7 Tree of Object-Oriented Parallel Programming Systems 33
Figure 5.8 Tree of Logic Parallel Programming Systems 33

Figure 6.1 Parallel Matrix-Vector Multiply Algorithm 34
Figure 6.2 Parallel Mergesort Algorithm 35
Figure 6.3 SMP Architecture 36
Figure 6.4 ccNUMA Architecture 37
Figure 6.5 Beowulf Cluster 38
Figure 6.6 Performance of MPI and OpenMP for Matrix-Vector Multiply . 39
Figure 6.7 Performance of MPI and OpenMP for Matrix-Vector Multiply, Logarithmic Scale 40
Figure 6.8 Performance of MPI and OpenMP for Mergesort on ccNUMA . . 41
Figure 6.9 Performance of MPI and OpenMP for Mergesort on ccNUMA, Logarithmic Scale 41
Figure 6.10 Performance of MPI on Beowulf with data size 1.000.000 42
Figure 6.11 Performance of MPI on Beowulf with data size 100.000.000 . . . 43

PARALEL PROGRAMLAMA SİSTEMLERİNİN SINIFLANDIRILMASI VE GRAFİK GÖSTERİMİ

ÖZET

Yüksek kazanımlı programlama olarak da bilinen paralel programlama, bir problemi daha hızlı çözmek için aynı anda birden çok işlemci kullanılmasına denir. Paralel programlama 1980'lerin sonunda popülerlik kazanmış, sürekli artan hız kazanma isteği ile popülaritesini arttırmıştır.
Günümüzde, ağır işlemler içeren birçok problem paralel olarak uygulanmaya çalışılmaktadır, buna örnek olarak nehir sularının simüle edilmesi, fizik veya kimya problemleri, astrolojik simülasyonlar verilebilir.
Bu tezin amacı, bilimsel hesaplama veya mühendislik amaçlı kullanılan yüksek kazanımlı yazılımları tartışmaktır. "Paralel programlama sistemleri" ile kastedilen kütüphaneler, diller, derleyiciler, derleyici yönlendiricileri veya bunun dışında kalan, programcının paralel algoritmasını ifade edebileceği yapılardır.
Yüksek kazanımlı program tasarımı için programcının dikkat etmesi gereken iki önemli nokta vardır: birincisi problemi iyi kavrayıp uygun bir paralel çözüm önermek, ikincisi ise doğru sisteme karar verebilmek. Doğru karar verebilmek için kullanıcının sistemler hakkında oldukça iyi bilgiye sahip olması gerekir. Bazen, birden çok yazılım ve donanımı bir arada kullanmak da gerekebilir. Programcı, problemi anladıktan sonra birçok sistem arasından birini seçmelidir, sistemlerin bazıları birbirleriyle yakından alakalı iken, bazıları tamamen farklıdır.
Bu tezde ilk olarak var olan paralel programlama sistemleri tanımlanır ve sınıflandırılır, bunun için güncel bildiriler esas alınmıştır. Özellikle algoritmik taslaklar ve fonsiyonel paralel programlama üzerinde durulmuştur. Ikinci olarak, güncel bilgileri depolamak ve bir kaynak yaratmak için wiki temelli bir web kaynağı oluşturulmuştur. Wiki tamamen dinamik, içeriği tüm kullanıcılar tarafından degiştirilebilen bir araçtır. Üçüncü olarak sistemlerin grafik gösterimini sağlayıp daha anlaşılır bir sınıflandırma yapabilmek için yeni bir sözdizimi tasarlanıp dinamik ağ çizebilecek webdot aracı ile bir araya getirilerek sistemleri temsil edecek ağı çizecek araç geliştirilmiştir. Bu sözdiziminin öğrenilmesi ve kullanılması son derece kolaydır ve wikide yorumların altına konduğundan bir karışıklığa sebep olmazlar. Bu sözdizimi ve araç sayesinde kullanıcılar kendi sınıflandırmalarını yapabilirler. Son olarak iki temel paralel programlama tipi, paylaşılan bellek ve mesajlaşma, iki farklı tipte algoritma kullanılarak karşılaştırılmıştır. Programlar OpenMP ve MPI ile gerçeklenmiştir, farklı paralel makinelerde koşturulup sonuçları karşılaştırılmıştır. Paralel makineler için Almanya'nın Aachen Üniversitesi'nin SMP ağı ve Ulakbim'in dağıtık bellekli paralel makineleri kullanılmıştır.

CATEGORIZATION AND VISUALIZATION OF PARALLEL PROGRAMMING SYSTEMS

ABSTRACT

Parallel computing, also called high-performance computing, refers to solving problems faster by using multiple processors simultaneously. Parallel computing became popular in the late 1980s and increased its popularity with the continual desire for more computing power.
Nowadays, almost every computationally-intensive problem that one could imagine, like the simulation of water levels in the rivers, chemical or physical problems, or astronomical simulations is tried to be implemented in parallel.
This thesis is aimed at discussing high-performance software for scientific or engineering applications. The term parallel programming systems here means libraries, languages, compiler directives or other means through which a programmer can express a parallel algorithm.
To design high performance programs, there are two keys for the programmer: The first is to understand the problem and find a solution for parallelization, and the second is to decide on the right system for the implementation, which requires a good knowledge about existing parallel programming systems. Sometimes, in a parallel application, several hardware/software tools are combined.
The programmer, after having understood the problem, has to choose between many systems, some of which are closely related, whereas others have big differences. To give an impression of the variety, a few systems are outlined here, others are explained in the rest of the thesis.

This thesis makes four contributions. First it describes and classifies existing parallel programming systems, thus bringing existing surveys up to date. Special emphasis has been given to skeletons and parallel functional programming. Second, it describes a wiki-based web portal for collecting information about most recent systems, which has been developed as part of the thesis. Wiki is a web engine that is fully dynamic and the content can be enriched by the users. Third, it reports on an extension of the wiki technology that has been introduced for the representation of the classifications. A special syntax and a visualization tool has been developed. The syntax in which users can add remarks to web pages, is easy to learn and use. The graph visualization tool uses the remarks to generate visual categorizations and clearly show relations between various systems. This syntax and tool allow users to have their own categorization scheme. Fourth, it compares two major programming styles message passing and shared memory with two different algorithms in order show performance

differences of these styles. Algorithms are implemented in OpenMP and MPI, performance of both programs are measured on the SMP Cluster of Aachen University, Germany and on the Beowulf Cluster of Ulakbim, Ankara.

1. Introduction

1.1. Motivation

Parallel computing, also called high-performance computing, refers to solving problems faster by using multiple processors simultaneously. Parallel computing became popular in the late 1980s and increased its popularity with the continual desire for more computing power.

Nowadays, almost every computationally-intensive problem that one could imagine, like the simulation of water levels in the rivers, chemical or physical problems, or astronomical simulations is tried to be implemented in parallel.

Parallel programming means dividing work into smaller pieces, distributing the pieces to different processors, and organizing communication. All of these activities require careful programming to achieve efficiency: how to divide the work, how to distribute, how to collect the results etc. Different paradigms can be adopted in the various stages: First, one can divide the problem into tasks and distribute it to the processors (task parallelism), or divide the data and let each processor do the same work on different data (data parallelism). Next, for the communication among processors, one can prefer a shared memory, where all processors can read or write data (shared memory programming), or pass messages between processors (message passing programming). The chosen paradigm depends on the particular algorithm and available machine.

This thesis is aimed at discussing high-performance software for scientific or engineering applications. The term parallel programming systems here means libraries, languages, compiler directives or other means through which a programmer can express a parallel algorithm.

To design high performance programs, there are two keys for the programmer: The first is to understand the problem and find a solution for parallelization, and the second is to decide on the right system for the implementation, which requires a good knowledge about existing parallel programming systems. Sometimes, in a parallel application, several hardware/software tools are combined.

The programmer, after having understood the problem, has to choose between many systems, some of which are closely related, whereas others have big differences. To give an impression of the variety, a few systems are outlined here, others are explained in the rest of the thesis.

OpenMP is an “application programming interface (API)” for Fortran and C++,

developed in the early 90's to be used for multi-threaded, shared memory programming. It is not meant to be used in distributed memory parallel systems. OpenMP is simple to use. The programmer inserts "OpenMP parallel directives" into performance critical sections of a sequential program until having obtained the desired speedup (OpenMP).
Message Passing Interface (MPI) describes a message-passing library, it is available for both Fortran, C and C++ programming languages. MPI is a standard for communication between processors working in parallel on a distributed memory system (MPIForum).
High Performance Fortran (HPF) is based on the procedural language Fortran, one of the oldest programming languages. There had been much research for parallelizing Fortran, but because of the portability problem, most of the parallel Fortran compilers couldn't survive. These works led to the High Performance Fortran Forum (HPFF), a coalition of industry, academic and laboratory representatives (HPFForum). HPF is a "data parallel language", this means that a single operation can be applied to different elements of a large data structure simultaneously, in HPF data structures are arrays.
Skeletons are reusable patterns, first introduced by Cole (1989) in his PhD thesis. Here is the definition of a skeleton by its inventor:

> A skeleton is a useful pattern of parallel computation and interaction which can be packaged up as "framework/second order/template" constructs (i.e. parameterized by other pieces of code), perhaps presented without reference to explicit parallelism, perhaps not. Implementations and analyzes can be shared between instances.

When deciding for a system, a programmer has to take different pros and cons into account, eg: (Leopold, 2001):

- Shared memory programming is easy to handle, programmer does not need to deal with data distribution, does not need to handle communication details.

- Data parallel programming is easy to handle, but data distribution plays an enormous role and only regular problems can be expressed easily.

- Message passing programming is harder for the programmer, but more efficient especially on a cluster, it may be necessary to use message passing.

- The use of skeletons requires the existence of appropriate skeletons in which the algorithm can be expressed (Rabhi and Gorlatch, 2002).

1.2. Contributions

This thesis makes four contributions. First it describes and classifies existing parallel programming systems, thus bringing existing surveys up to date. Special emphasis has been given to skeletons and parallel functional programming. Second, it describes a wiki-based web portal for collecting information about most recent systems, which has been developed as part of the thesis. Wiki is a web engine that is fully dynamic and the content can be enriched by the users. Third, it reports on an extension of the wiki technology that has been introduced for the representation of the classifications. A special syntax and a visualization tool has been developed. The syntax in which users can add remarks to web pages, is easy to learn and use. The graph visualization tool uses the remarks to generate visual categorizations and clearly show relations between various systems. This syntax and tool allow users to have their own categorization scheme. Fourth, it compares two major programming styles message passing and shared memory with two different algorithms in order show performance differences of these styles.

1.3. Outline

The thesis consists of six parts. Chapter 2 is an outline of parallel programming and existing classifications. Of course, not all aspects of these topics can be covered in full detail here, since this would fill several books, so only the ones with a high relevance are examined. Pointers to additional literature have been added for further reading. The focus of chapter 3 and chapter 4 are on new research about parallel systems and on the wiki classification, that has been developed in the thesis. In particular, chapter 4 describes new systems and explains reasons for classification. Chapter 5 gives some information on the implementation of the wiki engine, and describes the visualization tool. Chapter 6 concerns comparison of two main parallel programming systems, MPI and OpenMP, and chapter 7 outlines possible directions and challenges for future work.

1.4. Disclaimer

Trademarks and brand names have been used without explicitly indicating them. The absence of trademark symbols does not infer that a name or a product is not protected. All trademarks are the property of their respective owners.

2. Parallel Programming and Existing Classifications

2.1. Parallel Programming

Parallel computing splits an application up into tasks that are executed at the same time to achieve efficiency. Thus, a task is a program or a part of a program in execution. Early parallel programming environments often required a particular architecture, and were difficult to use. Nowadays, with the improvement of technology and development of new systems, parallel programming environments became less architecture-specific, more close to each other and easier to use.

2.1.1. Motivation for Parallel Programming

According to Leopold (2001), reasons for parallel programming are absolute performance, modeling, von-Neumann bottleneck, availability and scalability.
Absolute performance: The most important reason for using parallel programming is the computing power for computationally-intensive problems. A given level of performance can be easier achieved by a parallel computer than by a sequential computer. This makes parallel computers cheaper.
Modeling: Parallelism is the best way for modeling some real-world systems in which different parts work in parallel.
von-Neumann bottleneck: This term denotes the fact that access time to memory may be the bottleneck of the performance. Parallel computing increases memory capacity and therefore may speed up applications.
Availability: Parallel working means "working as a team", and if one of the components breaks, another one can take over its functions.
Scalability: Parallel systems are scalable, that is more components can be added.

2.1.2. Designing Parallel Programs

Designing parallel programs is much more complex than designing sequential programs. The programmer has to answer the following questions:

1. How to decompose the problem into subproblems?
2. How to distribute the data?
3. How to communicate between the processors?
4. How to synchronize processors?

Different parallel programming systems answer these questions differently, or put emphasis onto different questions. We consider these differences as important criteria for our classification. In the following, various approaches to answering the questions are described in more detail.

The decomposition of the problem can be classified as recursive decomposition, data decomposition, exploratory decomposition, and speculative decomposition (Grama et al., 2003). Recursive decomposition is suited for problems that can be solved with divide-and-conquer strategy. Subproblems are recursively decomposed into smaller subproblems. Data decomposition is suited for working on large data sets. Operations performed on decomposed data sets are usually similar. Exploratory decomposition is is used when the problem is searching a space of candidate solutions. Parallel tasks run until a desired solution is found. Speculative decomposition is used when, during the lifetime of the program, the next step depends on the the output of the previous one. The decomposition step determines the "degree of concurrency", that is the maximum number of tasks that can run in parallel. Often, we use a mixture of the decomposition techniques in different stages of the program.

Different approaches to decomposition can be classified as data parallelism and task parallelism. Data parallelism is applying identical operations to different elements of a large data structure. Task parallelism is applying different operations to the same or different data. Task parallelism is mostly preferred when the data set is large as compared to the amount of computations.

Like the decomposition step, the mapping step has much impact on efficiency. A good mapping scheme should exploit the parallelism that has been identified in the decomposition step. Moreover, it should avoid communication by mapping independent tasks onto different processors and related tasks to the same processor.

Communication between parallel processing elements is one of the major overheads in parallel programming. There are two main techniques for process communication: shared address space (shared memory) and message passing. Shared memory communication is easy to handle. Shared memory operations are simple read and write operations. The main problem of shared memory is data race. A data race occurs when several processing elements try to access the shared memory and at least one of them tries to write. Data races are a problem since the same program can yield different outputs.

Message passing is done by sending and receiving messages, it is harder for the programmer who has to be very careful about every detail. Usually, message passing is preferred when there is no physical shared memory. Some parallel programming systems like Linda implement a shared memory on top of a

physically distributed memory.
Communication can be synchronous or asynchronous. In synchronous communication, a common timing signal is established that dictates when the communication can occur. In asynchronous communication, messages may be sent and received at a different time, without synchronization.

2.2. Existing Classifications

Two alternative classification schemes are outlined here: one from the book of Leopold (2001)'s, and one from D'Ambra et al. (2002).

2.2.1. Classification by Abstraction Level and Communication Model

Leopold's classification is according to the abstraction level first, and second according to the communication type. The abstraction level determines how much the programmer is involved with parallel programming details. And then the communication model determines the dynamism of the systems. The classification is not strict, the book gives rather a loose framework for presentational purposes, discussing together related ideas. The following classes of parallel programming systems are distinguished:

2.2.1.1. Data-Parallel Systems

As mentioned before, data parallelism is applying the same operation to the elements of a large data set. Two types of data parallelism are distinguished: SIMD parallelism and SPMD parallelism. Parallel operations can be elementary or complex, the dataset can be a simple array, a set of data or a list of data.
SIMD parallelism uses simple operations at data-parallel steps, some systems are C*, MasPar Language (MPL), Parallaxis.
Data parallelism on arrays uses complex operations at data-parallel steps, an example system is High Performance Fortran (HPF).

2.2.1.2. Shared memory Systems

Shared memory programming uses task-parallelism. Multiple tasks run in parallel and communicate by reading from and writing to a shared memory. Shared memory programming is simple for the programmer because a specific memory organization is not needed and he/she doesn't need to be involved in the distribution of the data or in the communication details. As SMPs have shared memory, they are suited for shared memory programming.
Thread sub-model uses threads as parallel processing units. Threads share global variables and don't share the local ones, they can carry out different programs,

which means task parallelism. Typical systems are libraries like the Pthreads Library (POSIXThreads) and Java Threads (JavaThreads).
Structured shared memory programming sub-model has parallel regions in which several threads exist, started by a master thread. The most famous system is OpenMP.
One-sided communication doesn't handle the communication implicitly, but the programmer has to manage shared memory allocation and communication between processes explicitly. Systems are parts of Message Passing Interface-2 (MPI-2) and Bulk Synchronous Parallel Programming (BSP) (Hill et al., 1997).

2.2.1.3. Message Passing Systems

Message passing programming supports both SPMD and MPMD parallelism. Processes communicate by sending and receiving messages. This model is preferred when the machine has not a physical shared memory. In this case it is more efficient, but error-prone and time-consuming. Message passing systems are Message Passing Interface (MPI) and Parallel Virtual Machine (Lane, 1995).

2.2.1.4. Coordination Systems

Coordination models separate the computational part of a program from the communication part. This separations helps to have more structured programs. The programmer expresses the computational part of the program with a conventional language and the coordination part with a coordination language. Communication can be realized via a shared-data structure like in Linda, IBM TSpaces (IBMTspaces) and Sun JavaSpaces, or via coordination channels like in the message passing model. The oldest system is Communicating Sequential Processes (CSP), another system is Occam.

2.2.1.5. Object-Oriented Systems

Object-oriented models integrate parallelism with objects. This model is suited for task-parallelism and mostly for distributed systems. Examples are CORBA, Java RMI and DCOM.

2.2.1.6. High-Level Programming Systems

These models use a high level of abstraction, the programmer need not deal with low-level details.
Automatic parallelization is the best way to be far from all parallelization details. The programmer gives a sequential program to a parallelizing compiler which transforms it into a parallel one. Such compilers are Paradigm, Polaris, SUIF.
Skeleton model gives the programmer the opportunity to use skeletons,

well-known parallel programming patterns such as pipeline or task pool. Systems having skeletons are Structured Coordination Language (SC), High-Order Divide-and-Conquer Language (HDC) and Pisa Parallel Programming Language (P3L).
Compositional models distinguish program components. A system is compositional if every component of the program can be combined with other program components. In this case, properties of program components are preserved. This model is suited for task-parallelism. Systems are Program Composition Notation (PCN), Opus and TwoL.
Functional programming model is based on functional programming languages, the execution order of operations is not specified by the program. Function parameters may be evaluated in parallel. Such languages are Glasgow Parallel Haskell (GpH), Eden, Concurrent ML, Erlang.

2.2.1.7. Logic programming

Logic programming naturally leads to parallelism. Some languages are Gamma and Distributed Oz.

2.2.2. Classification of Advanced Environments

D'Ambra et al. (2002) distinguish advanced environments into two main classes: programming environments (PEs) and problem solving environments (PSEs). The term advanced environment stands for a conventional arrangement of both hardware and software resources to develop high-performance applications. Definitions of PEs and PSEs are given by the authors:

> A programming environment provides all the tools needed to design, code and debug parallel and/or distributed applications, according to a given programming model or language.

> A problem solving environment provides a set of user-friendly mechanisms and tools that allow to build-up an application, within a specific application domain, by gluing together, with an intuitive compositional model and using some kind of problem-oriented language, different building blocks.

A PSE enables its users to develop applications without having specialized knowledge of the hardware or software. Classification of PSEs is beyond the scope of this thesis, more detail can be found in (Problem Solving Environments Home Page).

2.2.2.1. Programming Environments

Traditional Programming Environments.

PEs have traditionally been developed as a sequential language with a communication library on top. A very famous and widely used PE is C/MPI. The parallelism is basically SPMD parallelism. Although efficients applications can be developed with C/MPI, traditional PEs are not considered user-friendly because they require very good knowledge of SPMD parallelism and the programmer has to deal with every detail. Other such PEs are C++/MPI, Fortran/MPI, C++/ACE.

Modern Programming Environments.

The common property of "modern" PEs is to free the programmer from parallelism details, giving the opportunity to inherit useful features.
Skeleton-based PEs like Pisa Parallel Programming (P3L) allow simple and concise code. The problem with these PEs is the limited number of skeletons.
Coordination languages are developed for problems having different software components which have to interact to perform complex tasks. Coordination languages sometimes need complex mechanism to integrate different sequential or parallel processes.
Design pattern-based PEs use design patterns, which are solutions for object-oriented programming to common problems in software design. Parallel design patterns are developed as solution to some parallel programming problems. Pros and cons are like skeleton-based PEs, one advantage is the use of object-oriented languages. The first design pattern-based PE is Correct Object-Oriented Pattern-based Parallel Programming (CO_2P_3S).
Component-based PEs are the most recent in high-performance computing. Components are developed for the purpose of reuse. With component technology, parallel code became usable outside the programming environment. Some component-based systems are CAFFEINE for developing SPMD parallelism and XCAT for grid applications.

3. New Research on Parallel Systems

Technology trends such as optical networking and web services suggest that parallel programming will increase in importance in the future (Foster, 2001). This thesis concentrates on the software side. In the first part of this section, we survey new trends in parallel programming. Therefore we give an overview of existing and recently suggested systems. For reasons of space and time, only part of the systems are included, with emphasis on systems that are used in practice.

3.1. Research Directions in Parallel Programming

The main goal of the new research is to ease parallel programming. The research concentrates on the management of shared memory, code mobility, reusable patterns and layered systems in which computation parts of the program are separated from the communication parts.

3.1.1. Code Correction and Debugging Tools for Systems Using Shared Memory Communication

Two main categories for this research are race detection and execution profile. As mentioned in chapter 2, data race is one main problem of shared memory systems, and data race detection is highly essential for debugging and assuring the correctness of these systems. An example detection tool is MultiRace, a tool for dynamic data race detection in multi-threaded C++ programs (Pozniansky and Schuster, 2003). A detection technique is Hybrid data race detection which is based on two old techniques, lockset-based detection and happens-before-based detection (O'Callahan and Choi, 2003). When a data race occurs on a shared memory protected by a lock, this is the violation of mutual exclusion. Lockset-based detection is a technique to detect these violations. If there is a data race between two events and we cannot say that one happens before the other one, this is a happens-before-based detection.
A program profile denotes the total count of basic block executions, cache misses etc. Developing the execution profile of parallel programs allow the programmer to study the shared variable data access patterns across threads and this is useful for decision of the architecture (Goel et al., 2003). An example of such tool is Whole Program Paths (WPP) which produces a single compact description of a

program's entire control flow (Larus, 1999).

3.1.2. Algorithm Design for Systems Using Shared Memory Communication

Parallel Random Access Machine (PRAM) is well known model for algorithm design. PRAM consists of processors having a very small local memory and accessing to a global shared memory. An example of new application is an NC algorithm for finding maximal acyclic set in any graph which is implemented on Exclusive Read Exclusive Write (EREW) PRAM (Windsor, 2004). Another important algorithm design scheme is the design of non-blocking algorithms (Doherty et al., 2004). Traditional approach is the use of locks to protect shared data. Problems of this approach are deadlocks and performance degradation. New approach is designing lock free algorithms. It is shown by Herlihy (1991) that it is possible to use some synchronization primitives.

3.1.3. Mobile Agents

In traditional models, each process is bound to a fixed location throughout its lifetime. Mobile agents are programs that can move through a network under their own control, migrating from host to host and interacting with other agents (Gray et al., 1997). Mobile agents are effective for distributed applications, they reduce remote memory access because they can move but they are more time-consuming than simple messages. Such example agents are MESSENGERS, Self-Migrating Threads and WAVE. MESSENGERS carry their own behavior through a network and perform computations at each node (MESSENGERS). Self-Migrating Threads are mobile threads which have the ability to move through a network designed in C++ and the ability to perform computations at each destination (Suzuki and Fukuda, 1999). WAVE is a system based on multiple "intelligent" agents which can process and communicate. Intelligent agents are coded recursively in WAVE language, they act like virus and self-spread in the network (WAVEGroup). For more information, readers can refer to (MobileAgents) and (Distributed Objects and Copmonents: Mobile Agents).

3.1.4. Design Patterns and Skeletons

The basic idea of using parallel design patterns and skeletons is reuse. While designing a complex program, it is more suitable for a programmer to reuse a solution of a similar program already realized instead of designing from the scratch. The definition of skeleton by its inventor was given in chapter 1, here is another definition by Bischof et al. (2003):

> Skeletons are reusable, parameterized components with well defined semantics and pre-packaged efficient parallel implementations.

A design pattern is a solution to common problems in software design and facilitates common structures existing in sequential object-oriented programming. The programmer chooses the appropriate design pattern from the pattern library. Parallel design patterns are parallel extensions of the design patterns. Skeletons and design patterns are very similar but different in the end, a skeleton is used for designing high-performance systems, a parallel design pattern requires other handling mechanisms such as fault tolerance, time lines and quality of service (Rabhi and Gorlatch, 2002). Most of the work on skeletal programming is based on functional languages as skeletons can be modeled as higher order functions (HOF)s and most of the work on parallel design patterns is based on object-oriented languages as a design pattern is an object-oriented approach. Systems using skeletons and parallel design patterns will be treated in the next chapter.

3.1.5. Layered Systems

These systems are sometimes called coordination systems or two-level systems. The separation of computation from the coordination makes two parts orthogonal to each other, so that a particular coordination style can be applied to any sequential language (Yang, 1997). These systems will be treated in the next chapter.

3.2. Overview of the Parallel Programming Systems

Parallaxis is a data-parallel language based on Modula-2, developed for SIMD computers. Parallaxis can be considered as low-level, the programmer has to specify number of processors, arrangement and connections between processors. It is application independent. For further information, see (Parallaxis).
High Performance Fortran. Fortran is one of the oldest programming languages developed for scientific computing and numerical analysis. There had been many researches for parallelizing Fortran but because of the portability problem, most of the parallel Fortran compilers couldn't survive. These works led to High Performance Fortran Forum (HPFF). HPF defines a set of extensions to Fortran90, a standard of Fortran released in 1990s. HPF applications are portable across platforms.
OpenMP is an API supporting multi-platform shared memory parallel programming in C, C++ and Fortran. An API is a collection of directive-based language extensions, runtime library routines and environment variables.

OpenMP is based on multiple threads working on the memory. It enables and simplifies code reuse, the parallelization is done by compiler directives. It has powerful lock mechanism to control critical regions. Recursive programming and loops with unknown number of iterations are the most important disadvantages of OpenMP. For further information see (OpenMP).

Cilk is a multi-threaded parallel programming language based on C. The runtime system is responsible of the communication and load balancing details which means that Cilk is implicitly parallel. Cilk uses "work stealing" scheme, the idle processors steal work (threads) from the busy one. Data race management is managed by locks but locks are dangerous with several processes, an alternative is guard statements (Cheng, 1997). A guard statement specifies which shared data to guard. Cilk is easy to program, the difference with C is that Cilk has more reserved words. Cilkchess is a very famous application of Cilk.

Unified Parallel C (UPC) is an extension of C programming language designed for high-performance computing on large scale parallel machines. UPC uses SPMD parallelism, the amount of parallelism is fixed at program's startup time. UPC views memory as a logically partitioned memory (Kuchera and Wallace., 2004). These partitions also are partitioned in two: shared and local parts. The local portion is accessed by the thread to which it belongs and the shared portion is accessed by all threads. There are many compilers for UPC like Berkeley UPC compiler, Compaq UPC compiler, GCC UPC compiler, and there is a benchmark designed to reveal UPC compilers performance weaknesses (El-Gahazawi and Chauvin, 2001).

Message Passing Interface (MPI) is a portable message-passing library available to both Fortran and C. Processes working in parallel but on different communicate by sending and receiving messages. MPI supports SPMD and MPMD parallelism types.

Intel C Compiler (ICC) supports SIMD parallelism and multi-threaded code development through auto-parallelism and OpenMP programming. The programmer doesn't have to manually insert OpenMP directives. It parallelizes automatically.

Pisa Parallel Programming Language (P3L) is an explicit parallel programming language developed at University of Pisa and the Hewlett-Packard Pisa Science Center. The Language is based upon skeleton-templates and allows parallel programs to be developed composing a small set of primitive parallel forms. The programmer must pay attention to the form of parallelism to be exploited, in return, P3L system handles lower-level parallelism details. P3L has two parts: sequential code and parallel code which is a set of skeletons. The sequential code is written in C programming language and the skeletons are written in a very similar

syntax to C programming language's syntax. P3L uses three types of skeletons: data parallel skeletons, task parallel skeletons and control skeletons.Data parallel skeletons define global operations over large data structures, where individual operations on single elements or substructures of the data structure are performed in parallel (Kuchen, 2002). Task parallel skeletons decompose tasks into subtasks which can be executed in parallel and pipelined in the end.

A Language With A Name (ALWAN) is a parallel language and programming environment developed at University of Basel, based on Modula-2 and (C or Fortran). The sequential (computation) and the parallel parts (coordination) of the program are separated which makes it a coordination language. It is a layered language: provides the programmer with high level constructs for the description of parallel coordination aspects (Hamdan, 2000).

Skeleton Imperative Language (Skil) is an imperative, C-based language enhanced with a series of functional features. It aims to provide a high programming level, which allows the integration of algorithmic skeletons. Current application of Skil compiler is the implementation of algorithmic skeletons for Parallel Adaptive Multigrid Methods.

Program Composition Notation (PCN) is a general coordination language influenced by the parallel declarative language Standard, developed at Argonne National Laboratory and the California Institute of Technology. Its major features are compositionality, determinism, implicit synchronization and higher-order functions. PCN provides a simple language for specifying concurrent algorithms, interfaces to Fortran and C, a portable toolkit that allows applications to be developed on a workstation or small parallel computer and run unchanged on supercomputers and integrated debugging and performance analysis tools.

Nested Language (NESL) is a data-parallel language based on ML. The basic idea of NESL is nesting data parallelism. Data parallelism in NESL is executed by operations over the data of same type. The main data parallel construct is "apply-to-each" which is a parallel construct applied to parallel sequence and which means parallelism in parallelism (Belloch, 1996). NESL is a strict language. In a strict language all function arguments are needed and they can be all executed in parallel. In a non-strict language, all of the function arguments are not needed which gives the obligation to determine expressions to execute in parallel (Hammond and Michaelson, 1999).

Streams and Iterations in a Single Assignment Language (SISAL) is a portable, high-performance, data-parallel functional programming language. SISAL code can be mixed with C or Fortran for hybrid applications. SISAL's parallelism is implicit, it uses control parallel loop constructs over arrays (Hammond and Michaelson, 1999).

MultiLisp is an extension of the functional programming language Lisp. MultiLisp includes constructs for causing side-effects. MultiLisp's constructs make the parallelism explicit. The parallel construct implicitly uses fork-join parallelism followed by a procedure call.

Simple CAml to MPI (Scampi) is a small library allowing functional programs written in Caml to make calls to MPI communication routines. It provides some MPI bindings and based on the static SPMD execution model: all processes are created at launch time and remain active until the end of the computation. Parallelism in Scampi is explicit.

Concurrent ML (CML) is a concurrent extension of Standard ML of New Jersey (SML/NJ). CML supports dynamic thread creation and synchronous message passing on channels. CML provides first class synchronous operations which are based on the notion of events as a first class data type and which comprises functions that produce base event values and combinators to combine event values into higher order operations (Hammond and Michaelson, 1999).

Concurrent Haskell is the concurrent extension of the lazy functional programming language Haskell. Concurrent applications are expressed explicitly. It adds two mechanisms to Haskell: processes and mechanism for process initiation and atomically mutable state support for inter-process communication and cooperation (Jones and A. Gordon, 1996). A debugger and a compiler exist for Concurrent Haskell: Concurrent Haskell Debugger and Glasgow Haskell Compiler.

A Symmetric Integration of Concurrent and Functional Programming (Facile) is a high-level, high-order programming language for systems that require a combination of complex data manipulation and concurrent computing. It is an extension of Standard ML. It is considered as a "reactive language", there are no clear notions of inputs and outputs or even of termination and the whole purpose of parallelism is to maintain a set of separate tasks interacting with an external environment (Ortega and Pena, 1998).

Eden is a declarative parallel functional language extending Haskell. Eden is explicit about process definition and implicit about process communication. Communication is asynchronous and is realized by message passing via communication channels. Eden is a layered language, it has two levels: communication and computation. Eden especially targets both transformational and reactive (concurrent) programs on distributed memory machines (Loidl et al., 2000).

Caliban is an annotation-based functional parallel language. Annotation property allows the programmer to partition the functional program and data amongst the computational resources available. Caliban is considered as a coordination

language which is very closely related to the functional language it controls and used to determine static mapping of parallel tasks to the processors (Hammond and Michaelson, 1999). The basic object in Caliban is a stream of values, a stream us communication link between computations (Taylor, 1993).

Glasgow Parallel Haskell (GpH) is a strict parallel functional language extending Haskell. GpH adds a primitive for parallel composition "par", that is used together with sequential composition "seq" to express how a program should be evaluated in parallel. The implementation of GpH is GUM. GpH uses annotation-based approach in which process creation, distribution etc. are under automatic dynamic control. GpH specifically targets transformational (parallel) programs on a range of tightly-coupled parallel architectures from shared memory to distributed memory machines (Loidl et al., 2000).

Higher Order Divide and Conquer Language (HDC) investigates the automatic parallelization of divide-and-conquer recursions. It is an extension of the pure and higher-order functional programming language Haskell. The only difference from Haskell is that HDC supports skeletons. An example of application is N Queens which is an important example of divide-and-conquer algorithm (Herrmann and Lengauer, 2000).

Parallelizing Using Farmed Functions (PUFF) is a compiler generating sequential Occam2 code from Standard ML and identifying useful parallelism in general linear recursion.The PUFF compiler relies on profiling information and performance modeling to determine which linear recursive functions should be implemented as processor farms. It uses the processor farm skeleton but doesn't support nesting of processor farm skeletons allowed in the system (Hamdan, 2000).

SkelML is a skeleton-based compiler for ML developed at Heriot-Watt University in 1994. SkelML compiler identifies useful parallelism in higher order function use and can transform prototypes to enhance the exploitation of parallelism through algorithmic skeletons. The compiler exploits the parallelism available in the program through a set of predefined skeletons. The skeletons are map, filter, fold, filtermap, mapfilter and foldermap. It uses automatic program synthesis to identify specific parallel patterns.

GoldFISh is a parallel version of FISh, designed for producing portable, implicitly parallel language. FISh supports both the functional and imperative programming in the style of an Algol-like language. GoldFISh is a purely functional language that will use shape analysis to determine costs, and hence appropriate distributions.Shape analysis is based on "shape theory" which considers that values associated with a data structure have a shape. Shape theory gives a precise categorical account of how data is stored within data structures,

or shapes (Barry Jay's Shape Theory Page). Fragments of GoldFISh programs are treated as FISh programs and compiled into simple, efficient, imperative code (e.g C or Fortran). GoldFISh acts like a coordination system, computation and communication are separated. GoldFISh supports skeletons.

EKTRAN is a vehicle for exploring skeletons nesting, based on a simple functional language influenced by FP, for coordinating skeletons through higher order functions (HOF). EKTRAN supports arbitrary nesting of map and fold. Programs written in EKTRAN are translated to Caml and the Camlot compiler for Caml is used to generate C code.

Parallel ML with Skeletons (PMLS) generates native code with skeletons from full pure-functional Standard ML program and supports static nested skeletons from nested higher order functions. It supports the full SML Core language and nested skeletons. Associated technology enables the automatic synthesis of higher order functions in programs that lack them, through proof planning. Static analysis and dynamic instrumentation, combined with performance models for skeletons, enable the identification of useful parallelism (Scaife et al.).

Concurrent Clean. Clean is a lazy, pure, higher order functional programming language with explicit graph rewriting semantics; one can explicitly define the sharing of structures in the language. Concurrent Clean has concurrency annotations to create functions which can be executed in parallel. Communication takes place automatically.A distinction has been made between parallel programs and concurrent programs in Concurrent Clean. Parallel programs have the same semantics as their sequential counterparts, so there is no explicit message passing and non-determinism. Concurrent programs have a different semantics, because explicit message passing and non-determinism are used. Both programming models are provided in Concurrent Clean (Serrarens, 1998).All objects are represented by graphs which makes Concurrent Clean suitable for the specification of process topologies (Hammond and Michaelson, 1999).

Irvine Dataflow (Id) is a non-strict, single assignment language and incremental compiler developed for MIT's Tagged-Token Dataflow Architecture planned to be used on Motorola's Monsoon Dataflow Multiprocessor (small shared memory multi-processors). In the dataflow model, remote requests are structured as split-phase transactions so that multiple requests may be in progress at one time (Hicks et al., 1993). Id is a layered language. Exploitation of inherent expressions, loop and function parallelism similar to SISAL's.

Parallel Haskell (pH) is a successor to the dataflow language Id and it adopts the notation and type system of the functional language Haskell. pH is fully parallel, and the approach it uses differs from other programming languages. For example,

the first and simplest programs that a pH programmer writes are parallel, and an advanced pH programmer learns judiciously to use explicit sequencing later (Nikhil and Arvind, 2001). It is possible to provide programmer annotations to indicate sub-expressions which should be evaluated in parallel but the annotations can only affect termination. pH includes all Haskell syntax, and the same type inference system.

eSkel is a skeleton Library for C programs supporting SPMD parallelism. The current implementation is very preliminary (Cole, 2004).

Skipper presents a skeleton-based programming technique for fast prototyping of reactive vision applications. Skipper consists of a skeleton library, compile-time system and a runtime system, its target is parallel C code (Sérot and Ginhac, 2002).

POOMA is high performance toolkit for scientific parallel computation developed in 1994 at Los Alamos National Laboratory to assist nuclear fusion and fission research. POOMA is based on C++ and supports data-parallelism and automatically parallelizes scientific computation. POOMA toolkit and a message passing library like MPI or Cheetah Message Passing Library automatically perform all computation and communication.

JOMP is an OpenMP-like API for parallel programming in Java (Bull and Kambites, 2000). It supports most of the OpenMP directives and uses fork-join parallelism. OpenMP directives are embedded in comments in the Java program. The program with the extension ".jomp" is compiled with JOMP compiler to the java source code. As Java applications are portable, so are JOMP applications. JOMP is a quite new system, it doesn't have many tools, it has an environment for the performance analysis and visualization of parallel applications written in JOMP (Guitart et al., 2001).

Parallel Sather (pSather) is the parallel version of the pure object-oriented language Sather. pSather presents a shared memory communication model and it supports both task and data-parallelism. The main idea in pSather is giving threads the ability to fork themselves and wait for a collection of threads to complete execution. pSather is simple and efficient.

Tübingen Parallel Objects (TPO) is a message passing library written in C++ on top of MPI. Its key features are easy transmission of objects, type-safety, MPI-conformity and integration of the C++ standard library (Grundmann et al., 2000).

Charm++ is an object-oriented, machine-independent parallel programming language based on C++. Charm++ programs run unchanged on MIMD machines and processes communicate with messages. There is a clear separation between parallel and sequential objects (Kale and Krishnan, 1993).

Jade is a Java-like language with message passing features. Communication occurs through asynchronous method invocation (DeSouza and Kale, 2003). Jade source is translated to Charm++ source which is than compiled and executed on the target machine. Jade is a quite new language, it doesn't support yet Java's standard libraries.

ARMI is a communication library that provides a framework for expressing fine-grain parallelism and mapping it to a particular machine using shared memory and message-passing library calls (Saunders and Rauchwerger, 2003). It is an advanced implementation of Remote Method Invocation (RMI) which is a communication model for object-oriented programs.

Lithium is a full Java library allowing parallel programs to be written and run according to the skeleton programming model on a network of Java machines. Lithium includes common skeletons such as pipeline, farm, map, reduce and divide-and-conquer.

Correct Object-Oriented Pattern-Based Programming System (CO_2P_3S) (pronounced as COPS) combines design patterns and frameworks in the object-oriented parallel programming programming domain. It supports Parallel Design Patterns (PDP)s. CO_2P_3S ensures correctness which ensures that once created, a parallel program has correct structure with all necessary parallelism details and openness which means that the programming system provides opportunities for performance tuning and allows the user to take full advantage of all language facilities and run-time libraries to improve the performance of an application. CO_2P_3S generates multi-threaded Java framework code for shared memory multiprocessor systems (MacDonald et al., 2002).

Structured Coordination Language (SCL) is a coordination language integrated with skeletons and combined with a base language and manages all parallelism aspects of the application. The base language cannot call SCL primitives (Darlington et al., 1995). SCL has two layers: primitive skeletons and data sharing skeletons. Primitive skeletons consist of array distribution, alignment of distributed arrays, data parallel primitives, computational primitives, and communication primitives. Data sharing skeletons manipulate shared elements.

Linda is a coordination language providing tuple space shared memory and is combined with a based language. Like SCL, Linda manages communication, it has four operations: put, remove, read and evaluate. Remove and read operations exist in two modes, blocking and non-blocking.

Gödel is a declarative, general-purpose programming language in the family of logic programming languages developed by Antony Bowers and John Lloyd at the University of Bristol and Pat Hill at the University of Leeds. Gödel supports infinite precision integers, infinite precision rationals, and also floating-point

numbers. It can solve constraints over finite domains of integers and also linear rational constraints. It supports processing of finite sets.

PARLOG is a logic programming language for parallel applications. Logic programming finds a solution to a given question by checking all of the given conditions. In PARLOG, finding a solution to each condition becomes a separate concurrent process. The shared variables of the conditions are the communication channels between the processes. PARLOG relations are divided into two types: single-solution relations and all-solutions relations (Clark and Gregory, 1986). Single-solution relation calls can be evaluated in parallel with shared variables acting as communication channels for the passing of partial bindings. Only one solution to each call is computed. All-solutions relation calls are evaluated without communication of partial bindings, but all the solutions may be found by an or-parallel exploration of the different evaluation paths.

Oz is a multi-paradigm language designed to support different programming paradigms: logic, functional, constraint, object-oriented, sequential, concurrent with equal ease. Oz is a concurrent object-oriented language, can be programmed in a very similar way to other such languages, like Java and Oz is a powerful constraint language with logic variables, finite domains, finite sets, rational trees and record constraints. It has a virtual machine which makes it portable. The Mozart system is an implementation of Oz and it provides state-of-the-art support in two areas: open distributed computing and constraint-based inference. Oz execution model is based on both concurrent logic programming and traditional search-based logic programming (Roy et al., 2003).

4. Classification of Parallel Systems

Classification is the act of distributing entities into classes or categories of the same type. Almost anything, animals, things, concepts, events etc. may be classified. Wikipedia introduces the term "taxonomy": classification is the act of placing an object or concept into a taxonomy (Wikipedia). Taxonomy may refer to either a hierarchical classification of entities, or the principles underlying the classification. We try to figure out a new classification of the old and new parallel programming systems.
The first classification takes as basis programming styles which mean programming languages systems are based on. Second classification is according to the implementation type of the systems. The third classification scheme combines both to show the wiki classification.

4.1. Classification according to the Implementation Type

As mentioned in chapter 1, parallel systems in thesis mean libraries, languages, compiler directives or other means through which a programmer can express a parallel algorithm. This classification takes as basis the implementation types.

4.1.1. Libraries

A library is a collection of subprograms used to develop software. Libraries cannot be executed independently, they define functions, routines or methods needed for the application. While using libraries, the programmer has to include the library to the imperative or object-oriented programming language. A compiler doesn't know what functions do, it only knows that there are some included functions, library calls cannot be optimized by compilers. Using libraries is easier to implement but applications are slower.
Libraries for parallel programming include: OpenMP, MPI, Scampi, JOMP, TPO, ARMI, Lithium, eSkel, Cheetah Messaging Library (only for POOMA) and MM Shared Memory Library (only for POOMA).

4.1.2. Languages

A programming language is a notation for expressing instructions to a computer and to develop applications. Codes written in a specific programming language

are translated to the executable codes by a compiler. Languages can be classified according to their level, a high level language is more user-friendly than a low level language.
A parallel programming language must provide support for the basic parallelism aspects: parallel execution, communication and synchronization. Most of languages used for parallel programming extend an existing sequential language.

- Parallel programming languages based on C: Cilk, UPC, Skil.
- Parallel programming languages based on C++: POOMA.
- Parallel programming languages based on Fortran: HPF.
- Parallel programming languages based on Modula-2: Parallaxis, ALWAN.
- Parallel programming languages based on Haskell: HDC, Concurrent Haskell, Eden, GpH, pH.
- Parallel programming languages based on SML: NESL, PMLS, CML, Facile.
- Parallel programming languages based on other languages: pSather (Sather), PCN (Standard), MultiLisp (Lisp), EKTRAN (FP), Concurrent Clean (Clean), GoldFISh (FISh), Charm++ (C++), Jade (Java).

Moreover, there are independent parallel programming languages that don't have a particular language as their root, but usually several. Such parallel languages include CO_2P_3S, P3L, SISAL, Caliban, Id, Gödel, PARLOG, Oz.
Finally, coordination languages like Linda, SCL can be coupled with C, Fortran or some other imperative or functional languages.

4.1.3. Compiler Directives

A compiler directive is a non-executable statement suggesting the compiler what to do, but which is not translated directly into executable code. Directives may be embedded within a program written in a base language. Compiler directives allow the programmer to use the same parallel code on both multi-processors and single-processor. If a compiler sees a directive that it doesn't understand, it can just ignore it. Another advantage is the incremental parallelism (Chandra et al., 2001).
Parallel systems based on compiler directives are OpenMP, HPF.

4.1.4. Parallelizing Compilers

A compiler is a program which translates the source code of a program into the executable code. Different compilers exist for different languages, mostly more programmers can find more than one compiler for the same language. A compiler can be preferred according to the application. Parallelizing compilers free programmer from low level parallelism details. They transform sequential program to parallel one and they are based on the parallelization of loops. Of course, parallelism must be understandable by the compiler.
Parallel compilers: ICC, PUFF, SkelML.

4.2. Classification according to the Programming Languages

This classification takes as basis, the programming language of the parallel programming system. We can say that parallel programming systems evaluation follows the one of the programming languages: from imperative or functional languages to the object-oriented or component-oriented which enable reuse. Most of the new systems allow reuse introducing design patterns or skeletons. We can distinguish systems with these historical approaches.

4.2.1. Parallel Programming with Imperative Languages

Imperative programming is a programming paradigm that describes computation in terms of a program state and statements that change the program state. Example languages are Fortran, Pascal, C and Ada.
With imperative languages, the programmer typically has to deal with low-level parallelism details and pay attention to the states: how to send messages, from where to where to send messages etc.
This class can be considered as "traditional". The most successful systems in this class refer to a communication library but compiler directives or extended languages exist as well. This class includes Parallaxis, HPF, OpenMP, Cilk, UPC, MPI, ICC, P3L, ALWAN, Skil, PCN, eSkel.
It is more convenient for this category to have a distinction between shared memory systems, data parallel systems and message passing systems. Table 4.1 gives an overview of imperative parallel programming systems.

Table 4.1. Imperative Parallel Programming Systems

system	implementation	parallelism	communication
Parallaxis	extends Modula-2	data parallel	-
HPF	compiler directives	data parallel	both
OpenMP	compiler directives	both	shared memory
MPI	library	task parallel	message passing
Cilk	extends C		message passing
UPC	extends C		shared memory
P3L	extends C	both	message passing
ALWAN	extends Modula-2		message passing
Skil	extends C		message passing
eSkel	library		message passing

4.2.2. Functional Parallel Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions. Example languages are Erlang, Haskell, Clean, Lisp and Caml. Contemporary functional languages have three key properties that make them attractive for parallel programming (Loidl et al., 2003):

1. Abstraction: Functional languages have two main abstraction mechanisms:

   - Function composition allows the composition of complex functions into simpler ones. This mechanism allows the decomposition of complex problems into simpler ones.
   - Higher order functions can take other functions as arguments and may also return functions as result.

2. Elimination of unnecessary dependencies: Functions map inputs to outputs, there are no other effect, thus functional languages doesn't contain side-effects which makes easier the detection and identification of potential parallelism. Functions are independent, all sequential dependencies are eliminated.

3. Architecture independence: Unlike imperative languages, functional languages enable a higher degree of abstraction over architecture characteristics through higher-order functions and polymorphism.

Parallel systems based on functional programming languages include NESL, SISAL, MultiLisp, Scampi, CML, Concurrent Haskell, Facile, Eden, Caliban,

GpH, HDC, PUFF, SkelML, GoldFISh, EKTRAN, PMLS, Concurrent Clean, Id, Parallel Haskell.

Most functional parallel programming systems emphasize ease of programming instead of higher performance (Leopold, 2001). Thus, functional parallel programming systems can be distinguished according to their level of abstraction. These systems contain different constructs for parallelism like skeletons and annotations. Figure 4.1 gives an overview of functional parallel programming systems classified according to Loogen (1999).

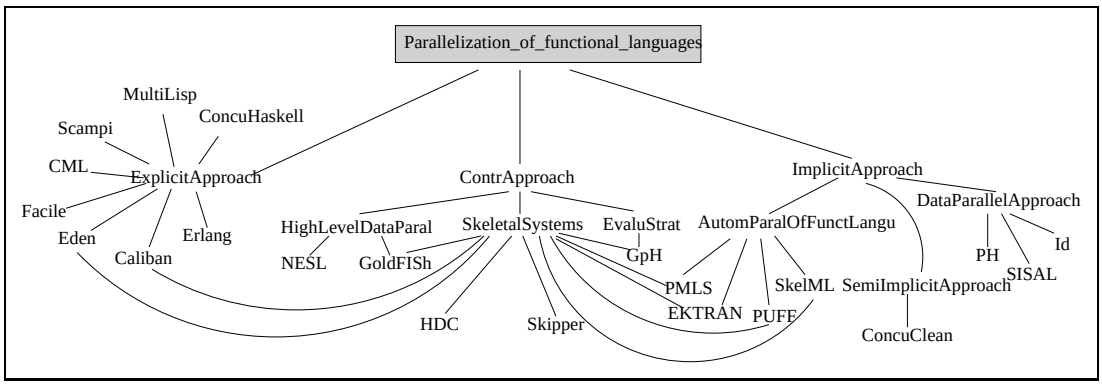


Figure 4.1. The classification of functional parallel programming systems is according to the abstraction level.

4.2.3. Object-oriented Parallel Programming

Object-oriented programming is a programming paradigm that has as basis objects as smallest units. Objects can perform computations and communicate with other objects. The support for encapsulation and software reuse of object-oriented programming languages using design patterns and frameworks allows programmers to write general application programs easily. A design pattern is a solution to common problems in software design and it facilitates common structures existing in sequential object-oriented programming. The idea of using object-oriented programming languages in parallel programming comes from the idea of reuse.

Parallel systems based on object-oriented programming languages include POOMA, JOMP, pSather, TPO, Charm++, Jade, ARMI, Lithium, CO_2P_3S. OpenMP and MPI exist for C++ as well, but the use is the same as imperative languages.

Like imperative parallel programming systems, most important aspect of object-oriented parallel programming systems is the communication.

Table 4.2 gives an overview of object-oriented parallel programming systems.

Table 4.2. Object-Oriented Parallel Programming Systems

system	**implementation**	**parallelism**	**communication**
POOMA	extends C++	data parallel	both
Parallel Java	extends Java	both	message passing
JOMP	library	fork-join	shared memory
pSather	extend Sather	both	shared memory
TPO	C++ library	task parallel	message passing
Charm++	extends C++		message passing
Jade	extends Java		message passing
ARMI	library		both
Lithium	library		message passing
CO_2P_3S	extends Java		shared memory

4.2.4. Parallel Programming with Logic Languages

Logical programming uses facts and rules to find a result to a given problem and offers some opportunities for implicit parallelism. A logic program can be represented as a tree. In sequential programs, nodes of a tree are visited in the predetermined order but in parallel systems, agents visit the tree nodes in parallel. There are two-parallelism types in parallel logic programming: "and-parallelism" and "or-parallelism". In these two parallelism types, conjunctions are evaluated simultaneously, and parent computations are blocked until their children have completed.

Parallel systems based on logic programming languages include Gödel, PARLOG, Oz.

Parallel programming with logic languages is a more theoretical subject for computer scientists. It is not widely used nor developed by many scientists. Table 4.3 gives an overview of logic parallel programming systems.

Table 4.3. Logic Parallel Programming Systems

system	**implementation**	**communication**
Gödel	mobile agents	message passing
PARLOG	mobile agents	message passing
Oz		message passing

5. Wiki Engine

A wiki server is a free tool for collaborative idea exchange and writing-informal, quick, and accessible. Wiki enables documents to be written collectively in a simple markup language using a web browser.
A page is represented in three ways in a wiki (Wikipedia):

- HTML code. This is the web page rendered by the web browser. When used alone, an HTML code is too complicated to allow fast-paced editing and distracts from the actual content of the pages.

- Source code which is editable by the users. The wiki server uses this code to produce the HTML code.

- Wiki text. Contents written in a simplified markup language.

Wiki implementation requires a web server and a database server. For the implementation, "Apache" "MySQL" have been used respectively. The wiki engine that was used for the tool is "MediaWiki". For the graph visualization a dynamic graph tool "Webdot" is used. Other tools can be used for different applications but the syntax would certainly be a little bit different.

5.1. The structure of the tool

Pages are parsed with PHP and the input file for webdot are also created by PHP. Wiki pages have names which represent them in the graph. A wiki page having as name "MPI" and describing MPI is represented with a node named as "MPI". A wiki page having as name "Message passing systems" is represented with a node named as "MessPassSyst". Long names are shortened for a clear view of the graph. When created, pages do not belong to any categorization graph. A "page" is added to a graph by a user. Many graphs are allowed for different users who are more interested in different aspects of parallel programming.

5.1.1. Syntax

The idea behind the graphical representation was to obtain a tree or mostly a graph. For this reason, systems represented by a wiki page have to be included in a graph giving the classification name which is the graph name and giving the descendants. The syntax is:

$< tree > tree_name[\#category_name]|[descendant_1, ..., descendant_n] < /tree >$

This syntax means that a page containing this syntax belongs to a graph having name "tree_name" and has as descendants descendant$_1$, ... , descendant$_n$ which exist most probably as "links" on this page. The category name and descendants are optional. A system can belong a graph without having any descendant or having many descendants. The "category_name" is used for a sub-categorization in the graph. The syntax is put under comments in wiki text by the user, it is not seen on pages.
The graph is a undirected graph or sometimes a tree, it can be converted to a directed graph with a very small syntax modification. Graphs are dynamic. The user of the wiki engine has only to include a system in a graph using the given syntax. When drawing the graphs, pages are parsed and every different graph is extracted. After the extraction, every graph's nodes and links are extracted. The cycles are not visited more than once, loops are eliminated with SQL queries. Parsing is done with PHP. Sources of functions for parsing and graph drawing are on enclosed CD.
The tree syntax must be compatible with the wiki's syntax. If there is a mistake, this is not considered fatal, the tree is drawn again but there is a message for the user telling the place of the syntax error.

5.1.2. Use of the Syntax

This part of the thesis describes how to use the given syntax with examples. Figure 5.1 is the main page of the parallel programming wiki portal.

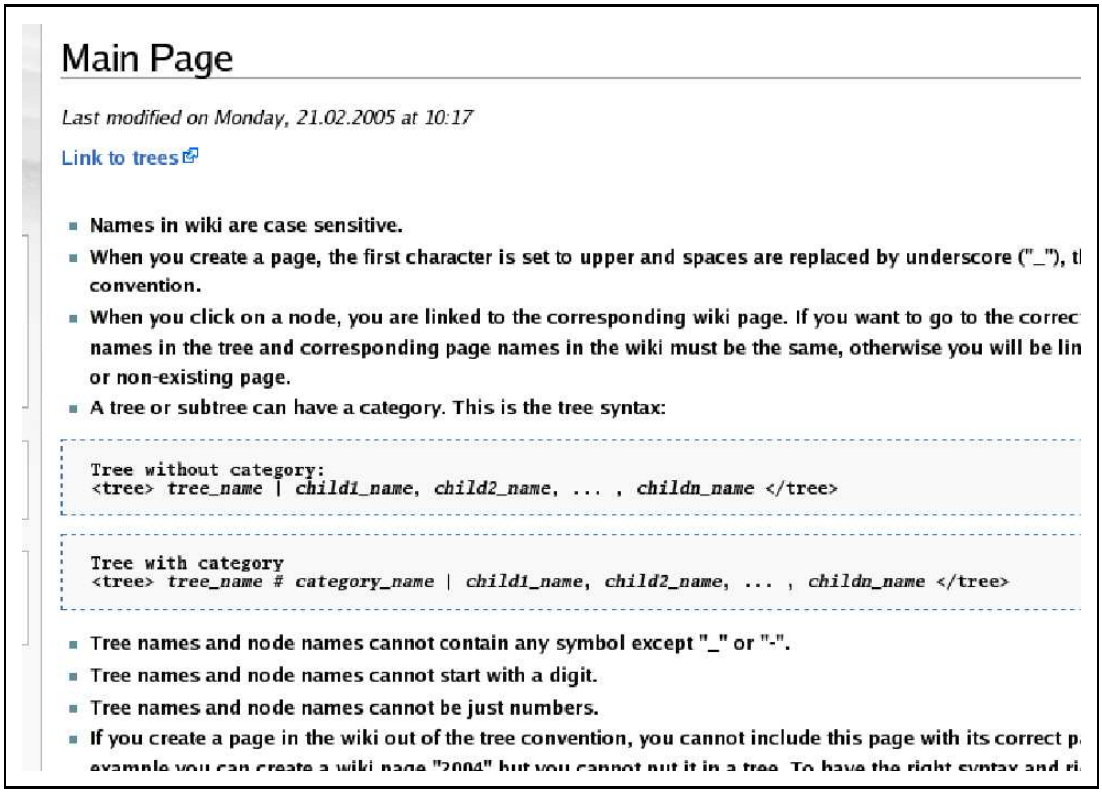


Figure 5.1. The contents can be changed by the users.

To add a new system to the wiki, we just “edit” the page and add a new link. In this editable page, all modifications are done with wiki’s own syntax. This syntax differs between wiki engines and their versions. The syntax can be learned from the user guide of the according engine. After modifying the contents, the page must be “saved” by clicking the save button. Figure 5.2 shows the editable page and Figure 5.3 shows new link.

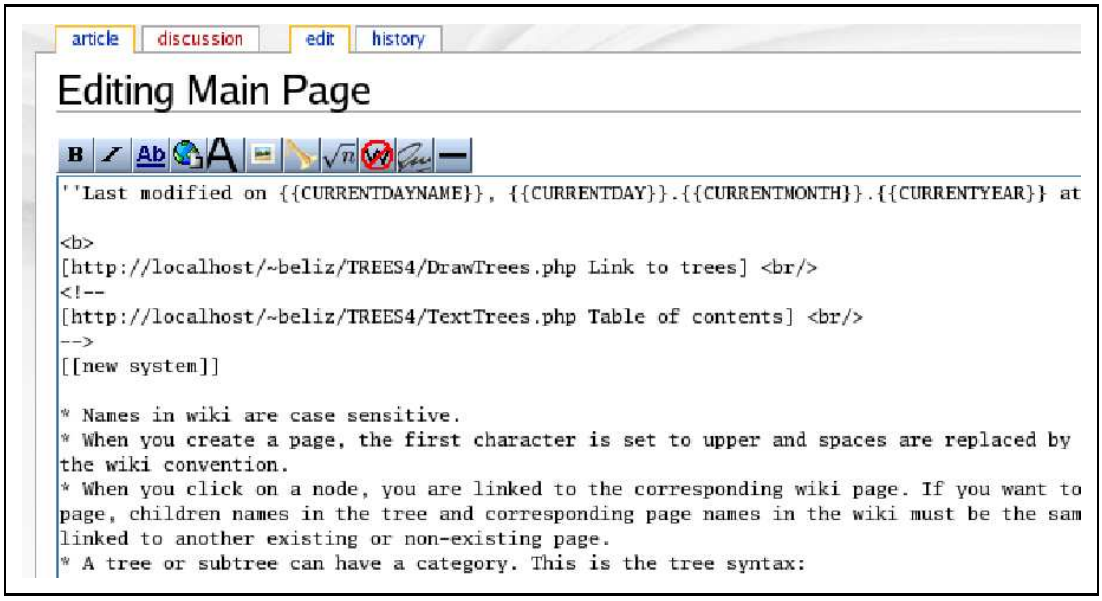


Figure 5.2. Users must use the wiki syntax of the current wiki version.

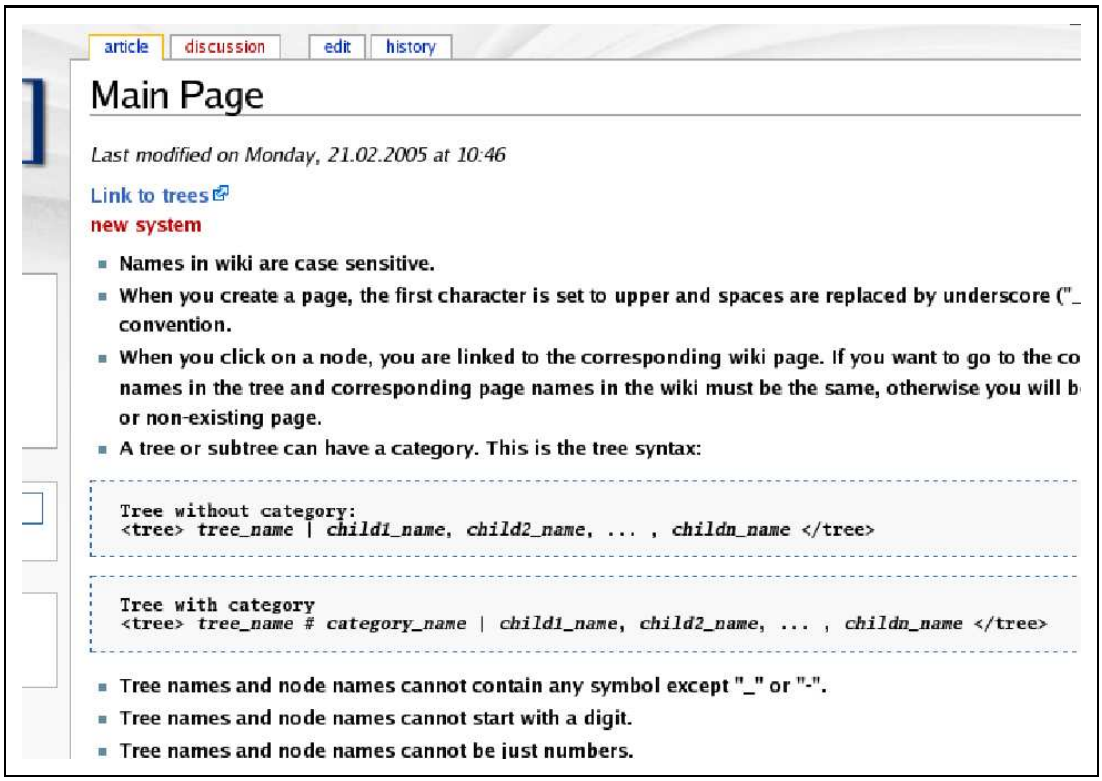


Figure 5.3. Added link appears on the page. An empty link is red, a full one is blue.

Now that the new system is added, we need to create a new subsystem and add this system to a tree. To put the tree syntax in comments is not an obligation

but it just allows a clearer view. Figure 5.4 shows the use of the tree syntax. The tree name indicates the tree to which "new system" belongs. Having children is not an obligation, the children part of the syntax could be blank.

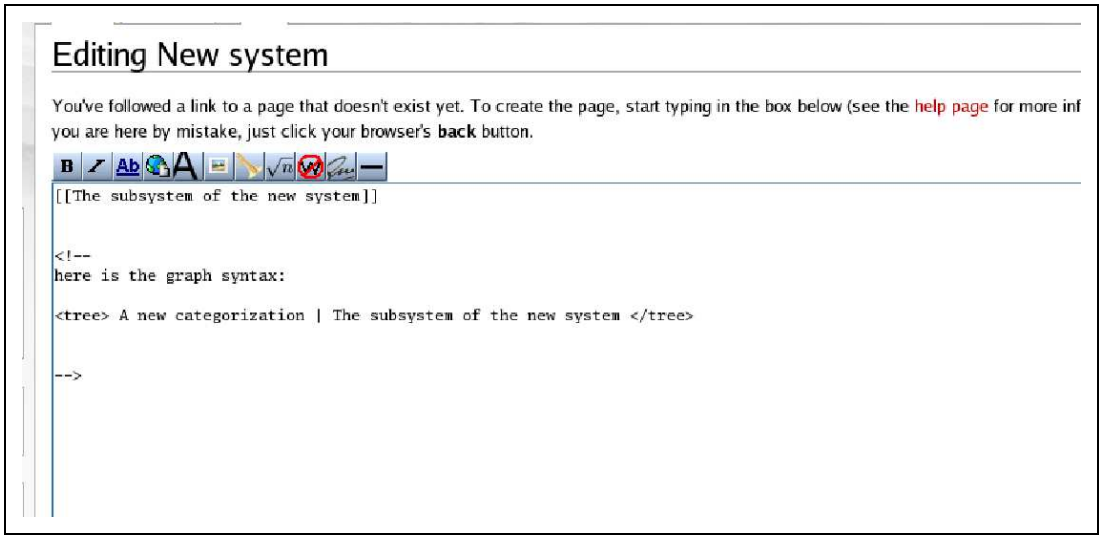


Figure 5.4. The content can be larger.

By following the link on the main page, users can observe the dynamic tree. The tree is dynamic which means that any user can edit the content of a node (a system) by clicking on the node and change the categorization, add or remove links. Figure 5.5 is a new categorization tree.

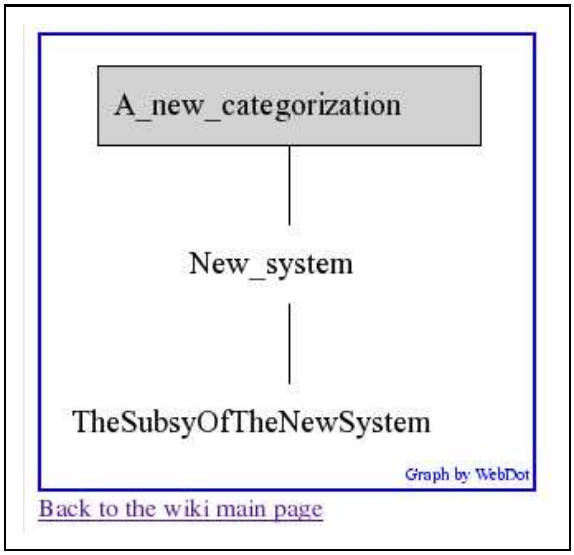


Figure 5.5. The new categorization tree with clickable nodes.

5.1.2.1. Sub-categorization

Subcategories are needed sometimes to indicate some points like library, language ... The syntax of a sub-categorized node of Figure 5.6 is:

$$< tree > ImperativeParallelProgrammingSystems\#compdir| < /tree >$$

5.2. Systems

If we try to do a whole classification like Skillicorn and Talia (1998) parallel systems look really complex. Instead of a large classification, smaller sub-classifications are preferred in this thesis.
As mentioned in chapter 4, the classification is based on the programming languages that is the core of the parallel programming system.

5.2.1. Imperative Parallel Programming Systems

Figure 5.6 shows imperative parallel programming systems tree.

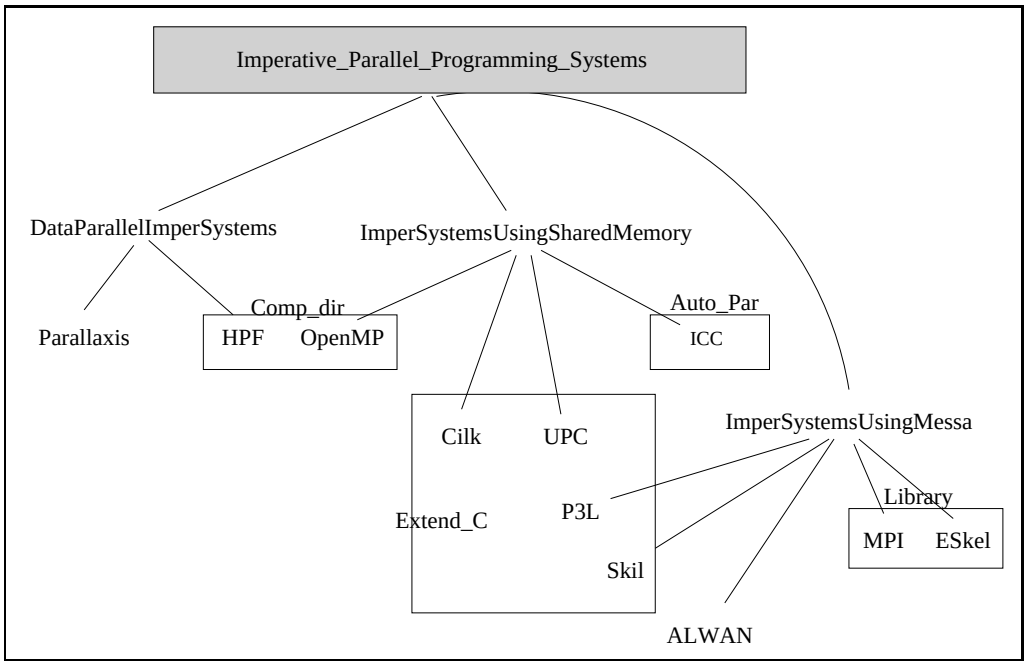


Figure 5.6. In this classification communication types are considered first, and a sub-classification is done according the implementation type.

5.2.2. Functional Parallel Programming Systems

The graph of functional parallel programming systems is given in Figure 4.1.

5.2.3. Object-Oriented Parallel Programming Systems

Figure 5.7 shows object-oriented parallel programming systems graph.

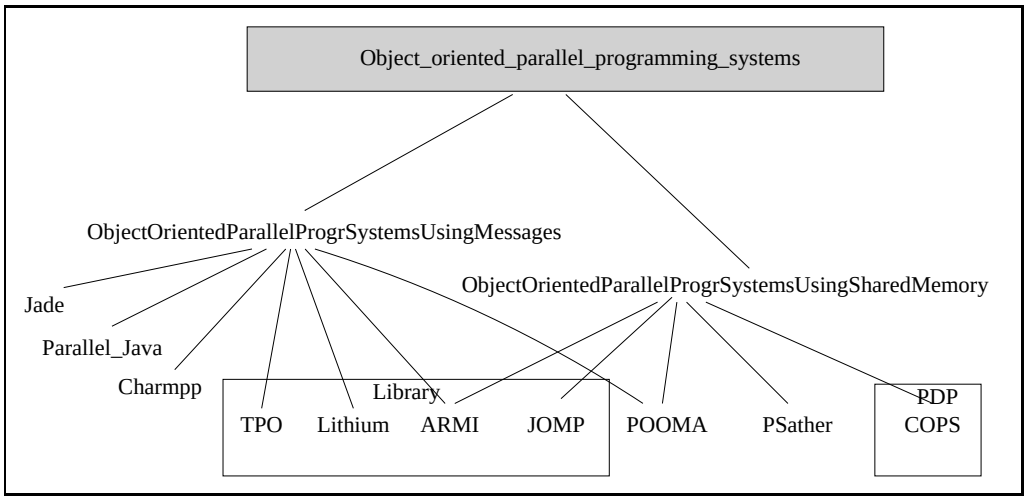


Figure 5.7. In this classification communication types are considered first, and a sub-classification is done according the implementation type. The only difference with imperative parallel programming systems is the parallel design patterns.

5.2.4. Logic Parallel Programming Systems

Figure 5.8 shows logic parallel programming systems graph.

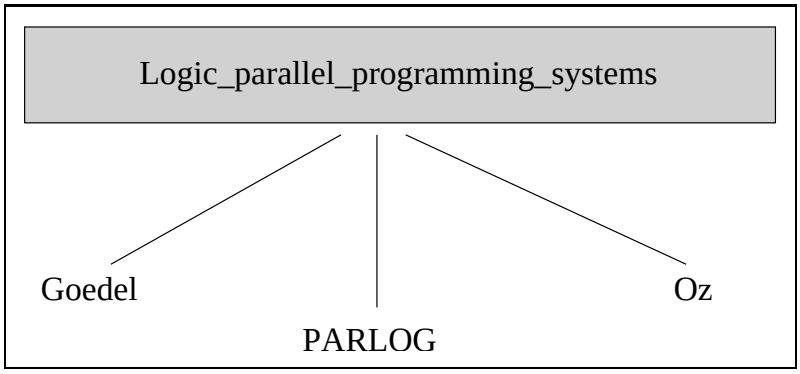


Figure 5.8. Logic parallel programming systems are nor widely used, they are more theoretical. A few systems can be cited as current.

6. Comparison of MPI and OpenMP

As mentioned in previous chapters, in parallel programming, the communication is via shared memory or message passing. Some hybrid approaches like "channels" exist as well but it is closer to shared memory as messages are put in a channel and the channel is shared. Most of the parallel systems (some are currently under development) target OpenMP, that is considered the core of shared memory programming or MPI, that is considered the core of message passing programming.
In this chapter, OpenMP and MPI performances are compared for two different types of algorithms, mergesort and matrix-vector multiply.

6.1. Algorithms

6.1.1. Matrix-Vector Multiply

Matrix-Vector Multiply is a "data parallel" algorithm. The main idea is to do the same operation on matrix arrays. Vector array is sent to each processor and arrays are distributed to the processors, multiplication is parallel, after the completion, each processor sends its result to the master processor. The source of OpenMP and MPI program's important functions can be found in enclosed CD. Figure 6.1 shows the matrix-vector multiplication algorithm.

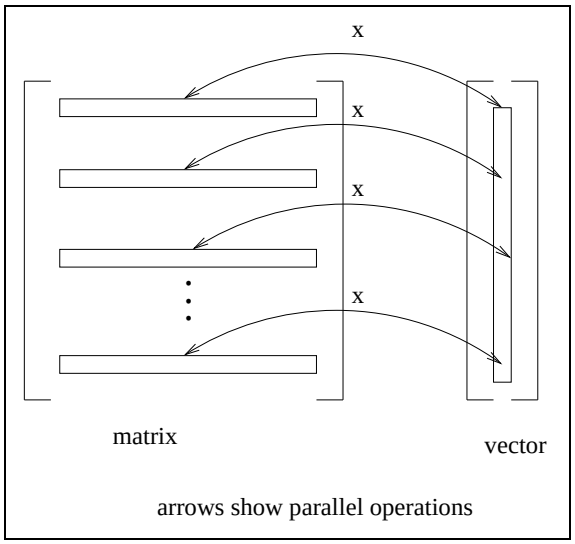


Figure 6.1. Matrix vector multiply. Multiplications are parallel.

6.1.2. Mergesort

Mergesort is a "divide-and-conquer" sorting algorithm. The main idea is to divide an array to sub-arrays and send the sub-arrays to the processors. Every processor does sorting first and after sorting, processors do merge in parallel. At each step, the number of processors decreases to its half. The source of OpenMP and MPI program's important functions can be found in enclosed CD. Figure 6.2 shows the mergesort algorithm with an example.

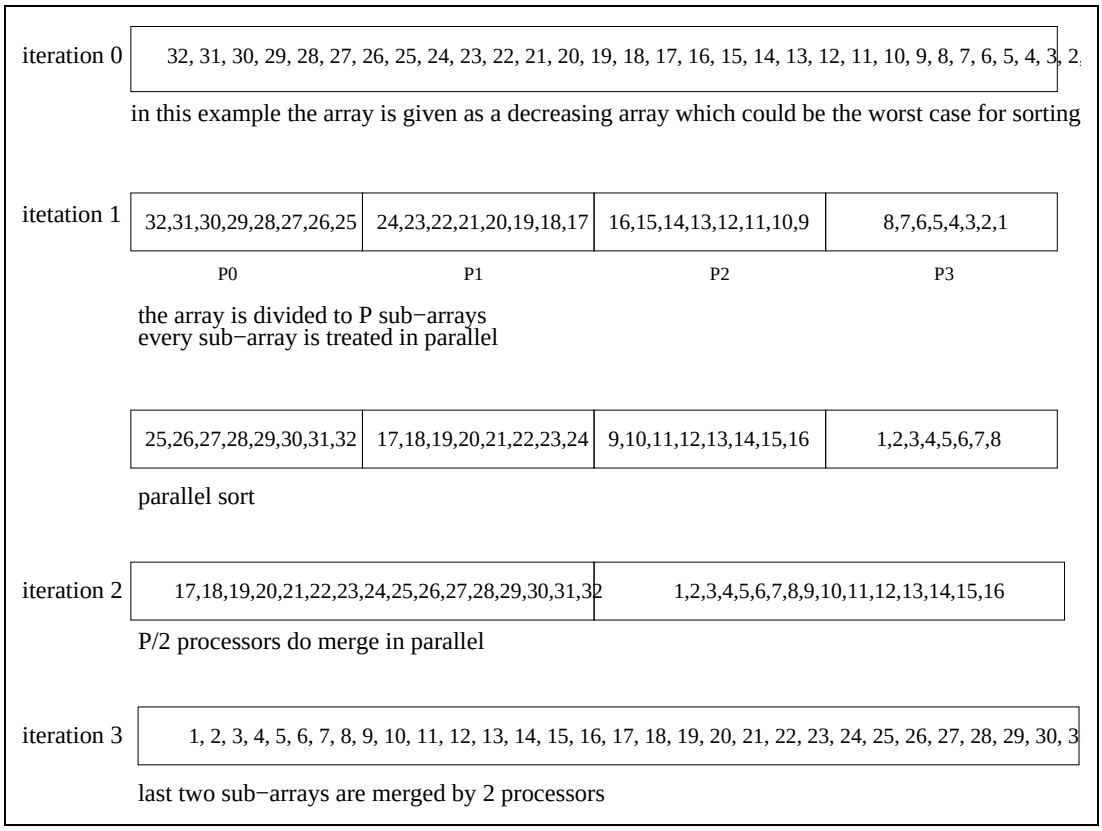


Figure 6.2. An example of mergesort algorithm with a decreasing array which is the worst case. Here, there are 4 processors, P=4.

6.2. Parallel Machines

6.2.1. SMP

In SMP architecture, each processor has access to each memory module which makes the memory "shared". The term symmetric means that for each processor, access time to each memory module is equal. For the programmer, SMP looks like

a sequential machine on which multiple processes run in a time-shared manner (Leopold, 2001). SMP is hard to scale because of the latency problem, accesses to memory modules increase with increasing number of processes. Figure 6.3 shows an SMP architecture.

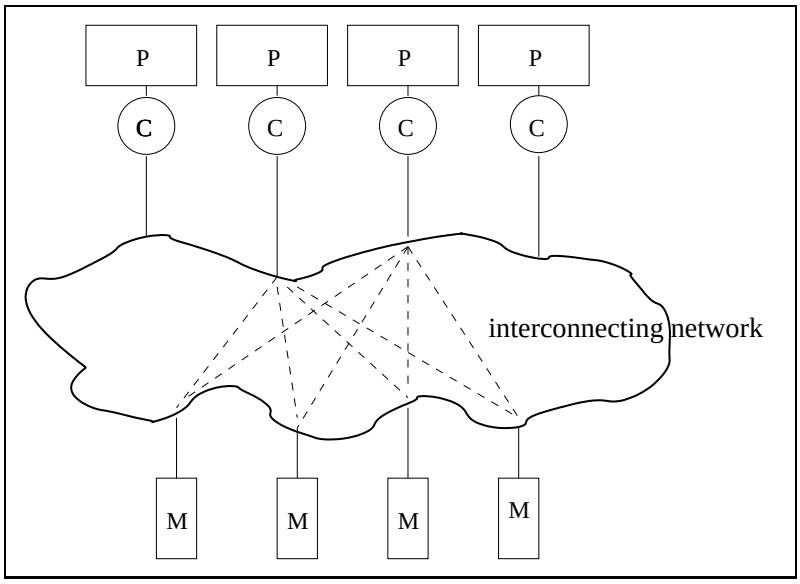


Figure 6.3. P means processor, C means cache and M means memory.

6.2.2. ccNUMA

Like SMP architecture, each processor has access to each memory module which makes the memory shared. The difference is that each processor is associated with a memory module directly, this module is called "local" and accesses to this local memory are faster than accesses to other memory modules. This difference destroys the symmetry and makes memory accesses "non uniform". ccNUMA architecture support complex protocols for cache-coherence which cause an overhead with increasing number of nodes. Cache coherency problems occur when cached data's copy's memory cell is written. Figure 6.4 shows ccNUMA architecture.

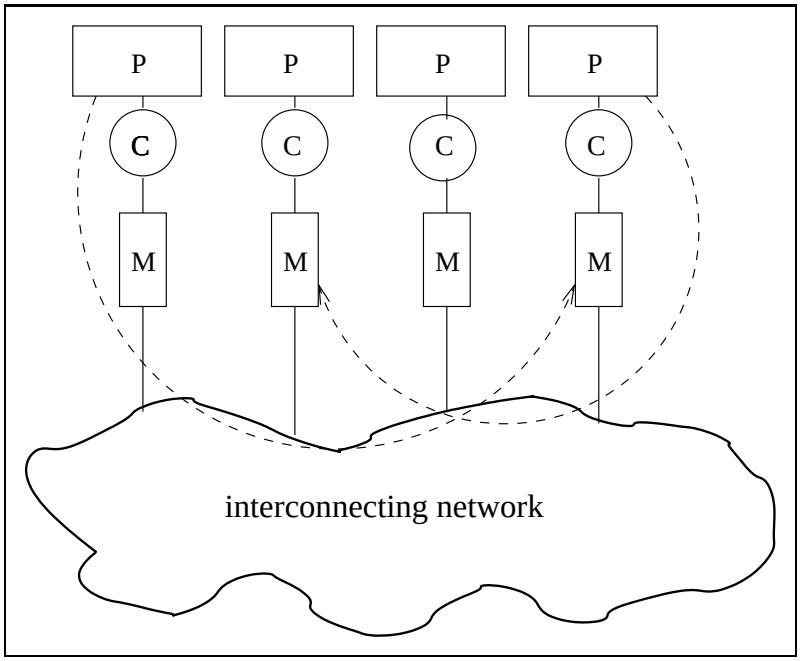


Figure 6.4. P means processor, C means cache and M means memory. C holds frequently accessed data.

6.2.3. Beowulf Cluster

A beowulf cluster consists of distributed memory parallel computers. Each computer is independent and support for cache coherence is lacking. The system has one access point and access to other computers occur from the access point. The user has to connect to every computer separately. Mostly, in beowulf clusters nodes run the Linux operating system. Beowulf clusters are easier to scale but harder to program. Most important advantage is lower cost. Figure 6.5 shows a beowulf cluster.

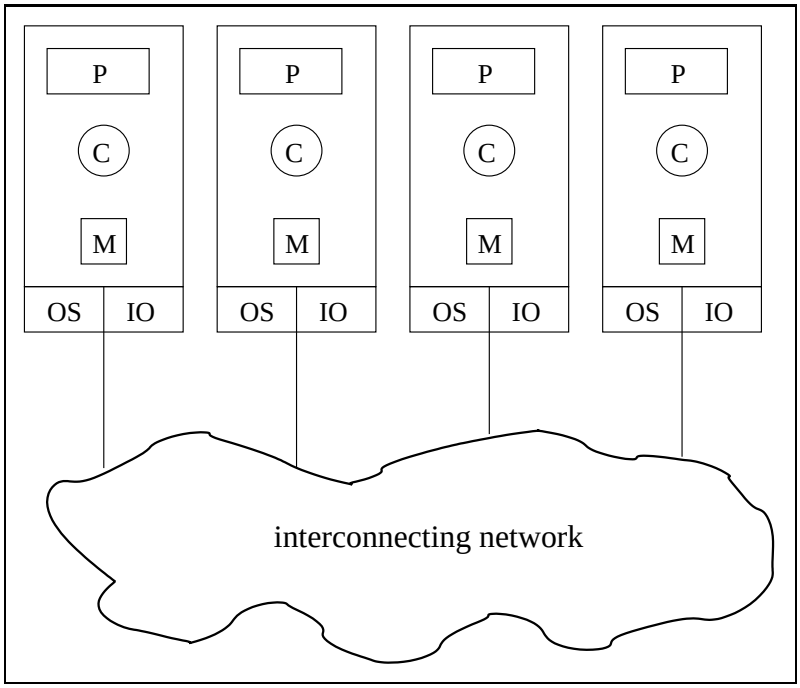


Figure 6.5. P means processor, C means cache and M means memory, OS means Operating System and IO means Input/Output. There is no shared memory module.

6.3. Performance Comparison

Programs are tested on Sun SMP-Cluster of University of Aachen and on Ulakbim's Beowulf Cluster.
SMP-Cluster currently consists of

- 16 nodes with 24 Ultra Sparc IV 1.2 GHz processors and 24 GB of shared memory each (SMP).

- 4 nodes with 72 Ultra Sparc IV 1.05 GHz processors and 144 GB of shared memory each (ccNUMA).

All 672 CPUs have a 900 MHz clock cycles and a total main memory capacity of 960 GB. All SMP compute nodes are connected to each other by Gigabit Ethernet.
Beowulf consists of a server node with 2 Intel Xeon 2.80 Ghz processors, 2 GB memory and 600 GB local hard disc and 128 computing nodes with Intel PIV 2.66 GHz processors, 1 GB memory and 80 GB local hard disc. All computing nodes are connected to each other by Gigabit Ethernet.
All nodes of both systems were not always usable by all the programmers. Mostly, using 128 processors was not possible, results for one essay are not considered,

only at least three computations are done to be sure about measured times. Algorithms are implemented with C and C++, OpenMP and MPI time functions are used to measure time. The measured time is the sum of computation and communication times. Largest data sizes that were allowed are used. For mergesort algorithm, three cases are considered: best case in which the array is already a sorted array, worst case in which the array is a decreasing array and a random array. For sorting sub-data, standard quicksort function "qsort" of C is used. Measured times for matrix-vector multiply are on Appendix A and measured times for mergesort are on Appendix B.

6.3.1. Matrix-Vector Multiply Algorithm

On SMP, OpenMP has a better performance than ccNUMA. As it can be seen in Figure 6.6, in matrix-vector multiplication OpenMP gave a much better result than MPI which has lower performance with increasing number of processors. Figure 6.7 gives the same graph in logarithmic scale for a better view.

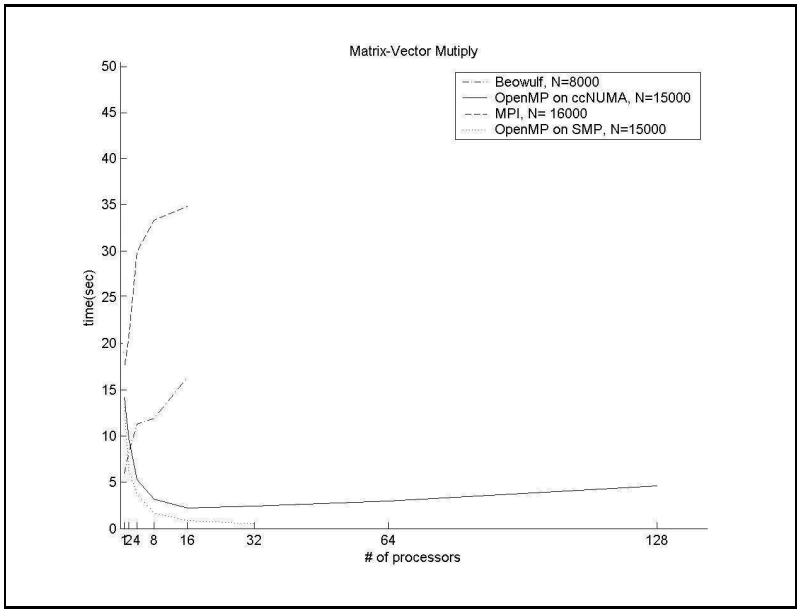


Figure 6.6. Solid line shows OpenMP on ccNUMA, dotted line shows OpenMP on SMP, dashed line shows MPI on ccNUMA, dashed and dotted line shows MPI on Beowulf.

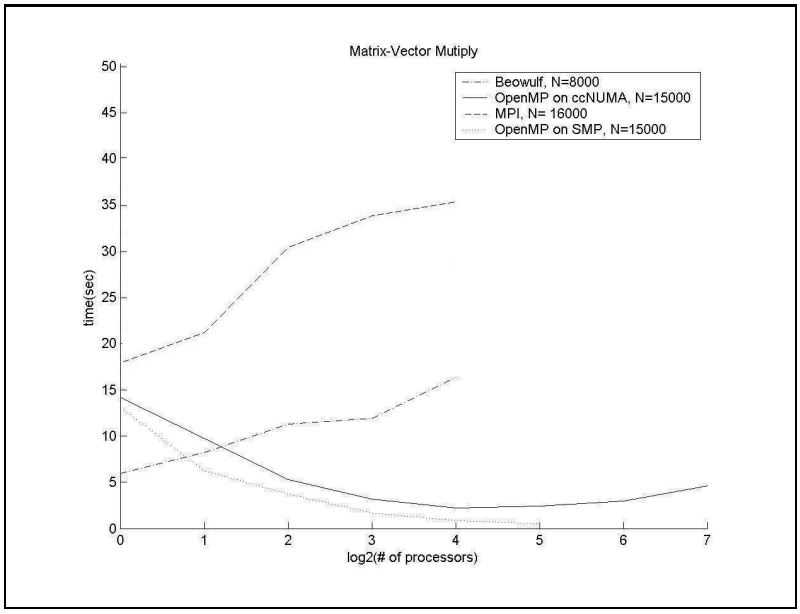


Figure 6.7. Logarithmic scale. Solid line shows OpenMP on ccNUMA, dotted line shows OpenMP on SMP, dashed line shows MPI on ccNUMA, dashed and dotted line shows MPI on Beowulf.

We can comment this result:

- SMP has a better performance than ccNUMA. But because it is harder to scale, SMP Cluster has less processors and we cannot test with more processors like in ccNUMA.

- In ccNUMA, after some processors (here 16) program time starts to increase with increasing number of processors which means that after 16 processors, communication cost increases and brings down the performance.

- MPI on ccNUMA and MPI on Beowulf have bad performance with increasing parallelism. This means that when implementing this algorithm with MPI, with this data size communication cost brings down the performance. This result doesn't mean that MPI cannot be used for this algorithm. If the additional tests with larger data size had been done before, we might have reached to the better performance. Working with larger dataset was not possible on neither of two systems because matrix size is N^2.

6.3.2. Mergesort Algorithm

For the divide-and-conquer algorithm, OpenMP and MPI have very close results on ccNUMA as can be seen in Figure 6.8 and Figure 6.9 which is the same graph

in logarithmic scale. On this system, results of best, worst and random cases were very close, so there no distinction has been made between these cases on this system.

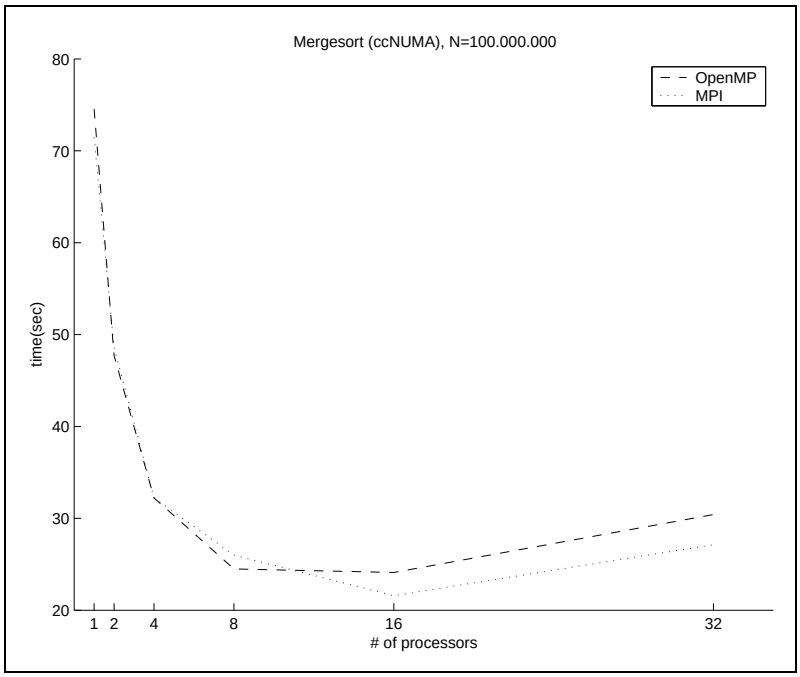


Figure 6.8. Dashed line shows OpenMP on ccNUMA, dotted line shows MPI and data size N=100.000.000 for both.

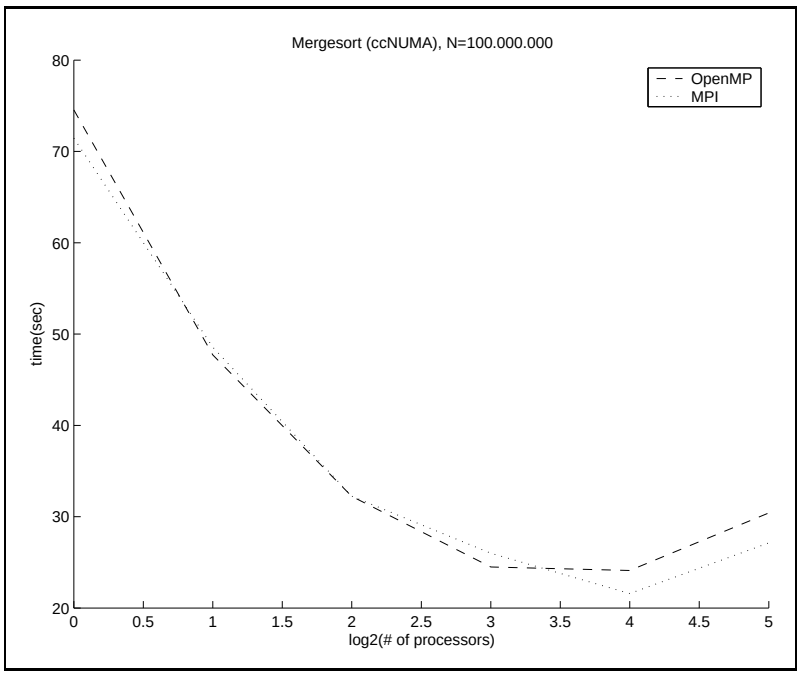


Figure 6.9. Logarithmic scale. Dashed line shows OpenMP on ccNUMA, dotted line shows MPI and data size N=100.000.000 for both.

We can comment this result:

- Up to 4 processors, MPI and OpenMP have very close results.
- Increasing number of processors from 4 to 8, OpenMP has better speedup.
- Increasing number of processors from 8 to 16, OpenMP's performance doesn't improve, MPI performance still improves.
- After 16 processors, communication cost brings down the performance for both OpenMP and MPI.

Finally, we can say that for less than 8 processors, OpenMP can be preferred with very small performance improvement. With more than 8 processors, MPI has better performance, improvement until 16 processors and worsen with more than 16 processors but still better than OpenMP.
MPI performance on Beowulf Cluster is a little bit different from the one on ccNUMA. All cases have different results that must be commented. Figure 6.10 shows MPI results of best, worst and random cases with data size 1 million and Figure 6.11 shows best and worst cases with data size 100 million. The random case for the latest didn't work because with random number generator the system had time out, this shows that this was the forcing data size.

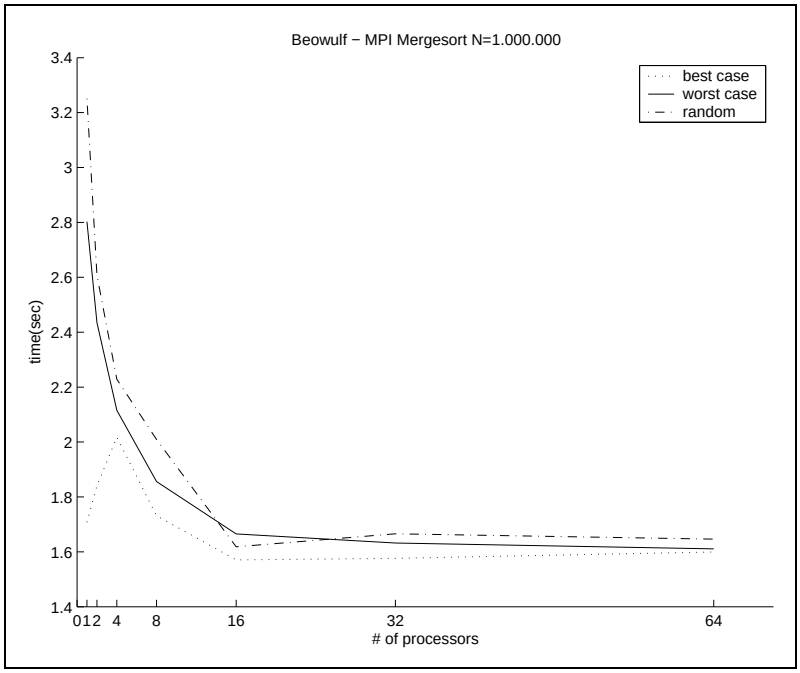


Figure 6.10. Dotted line shows best case, already sorted data, solid line shows worst case, decreasing data, dashed line shows random data.

Mergesorting in random and worst cases has very close results. As expected before, execution time decreases with increasing parallelism. The unexpected result is the best case where the execution time increases from 1 processor to 2. When there is only 1 processor, the program is a sequential program, data is sorted using C's qsort and until more than 8 processors, it is faster than dividing, sending and collecting data. With more than 8 processors, the result is like it was expected before. More than 16 processors cause high communication cost and there is no more performance improvement with more than 16 processors. Lack of the same attitude on SMP cluster could be due to the larger data size.

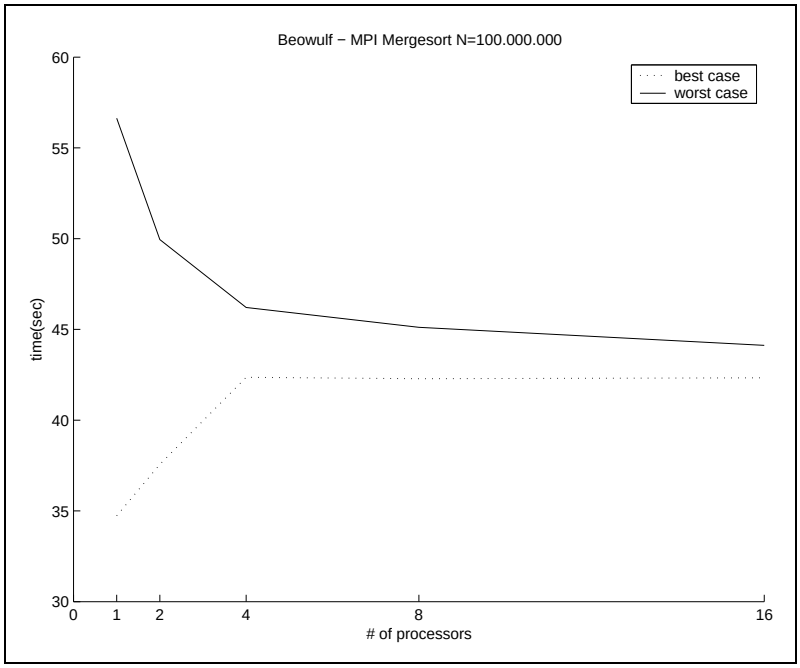


Figure 6.11. Dotted line shows best case, already sorted data, solid line shows worst case, decreasing data, dashed line shows random data.

Like the previous case, for the worst case bigger dataset has the same characteristics but for the best case it has the same only at the beginning. In the best case, parallelism doesn't improve performance, nor brings down it. This must be due to the overload of the whole system with such a big data size.

6.3.3. OpenMP vs MPI

Users must take in account the application and pros and cons of both OpenMP and MPI and decide which system to use. But one thing to not forget is that finding a shared memory machine is a very hard task. OpenMP is easier to program because the programmer need not be divided and distributed to

processors. Parallelizing a sequential code is easier for OpenMP, it can be done step-by-step which is called “incremental parallelism”. The programmer must pay attention to synchronization for both, but synchronization errors occur more in shared memory. In return, message passing is harder for the programmer and has higher performance.

Before having to choose between OpenMP and MPI, programmers have to decide about whether or not using parallelism. The results of matrix-vector multiply algorithm show that even with such big data set parallelism doesn’t have any advantage. Another architecture type could be developed for implementing such data-parallel algorithms. The mergesort algorithm had a little higher performance with MPI, systems having “messaging skeletons” or “messaging patterns” could ease the programming.

7. Conclusion

7.1. Summary

The main target of this thesis was overview research on parallel programming systems and make a clear categorization to guide users of parallel programming systems. To accomplish this task, a graph visualization tool is integrated with a wiki engine which is filled with information about systems being part of current research area. In order to realize the graph visualization tool, a syntax easy to use and learn is developed. For instance, wiki engine and graph visualization tool work independent from each other because in case of an upgrade, dependence could have caused problems.
The second target was to compare two parallel systems with two different types of algorithms. OpenMP and MPI had been chosen and programs had been run on parallel machines. There were two reasons for choosing OpenMP and MPI: first, most of the systems target OpenMP and MPI's performances and many systems are developed to support OpenMP or MPI, sometimes even target directly to have C/MPI or C/OpenMP code, second, OpenMP and MPI are the most common parallel programming systems and parallel machines mostly have support for just OpenMP and MPI for the users. Having access to parallel machines was the most difficult part of this thesis. Even having access was not always enough, having rights or priority was impossible.

7.2. Outlook

Improvements and further work could be done on several parts of the thesis. As mentioned in previous chapters, new research is mostly for easy programming and the classification of this thesis was in the point of view of a programmer. The wiki engine and the visualization tool could guide anyone interested in parallel programming. Many other classifications having different criteria (performance, ease of programming, application area, ...) could be done and could present parallel programming users new approaches and give new ideas.
Another important point is comparison. Comparison of all systems could be done. A good idea would be to compare HDC, OpenMP and MPI with a divide-and-conquer algorithm or add different kind of skeletons to a skeletal systems and compare them with still OpenMP and MPI. After this research,

testing several systems on several parallel machines could suggest architectural improvements.

REFERENCES

Barry Jay's Shape Theory Page. http://linus.socs.uts.edu.au/ cbj/.

Belloch, G. E., 1996. Programming Parallel Algorithms. *Communications of the ACM*, **39**.

Bischof, H., Gorlatch, S. and Kitzelmann, E., 2003. Cost Optimality and Predictability of Parallel Programming with Skeletons. *Euro-Par 2003, LNCS 2790*, pages 682 – 693.

Bull, M. and Kambites, M., 2000. JOMP - an OpenMP-like Interface for Java. *ACM Java Grande Article*.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R., 2001. Parallel Programming in OpenMP. Morgan Kaufmann.

Cheng, G.-I., 1997. Algorithms for Data-Race Detection in Multithreaded Programs. *PhD thesis*, Massachusetts Institute of Technology, Massachusetts, USA.

Clark, K. and Gregory, S., 1986. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

Cole, M., 1989. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press and Pitman.

Cole, M., 2004. Bringing Skeletons out of the Closet. *Parallel Computing*, **30**, 389 – 406.

D'Ambra, P., Danuletto, M., di Serafino, D. and Lapegna, M., 2002. Advanced environments for parallel and distributed applications: a view of current status. *Parallel Computing*, **28**, 1637–1662.

Darlington, J., Guo, Y., To, H. W. and Yang, J., 1995. Parallel Skeletons For Structured Composition. *ACM Principles and Practice of Parallel Programming, PPoPP'95*.

DeSouza, J. and Kale, L. V., 2003. Jade: A Parallel Message-Driven Java. *Proceedings of the 2003 Workshop on Java in Computational Science, held in conjunction with the International Article on Computational Science (ICCS 2003)*.

Distributed Objects and Copmonents: Mobile Agents. http://www.cetus-links.org/oo-mobile-agents.html.

Doherty, S., Detlefs, D. L., Grove, L., Flood, C. H., Luchangco, V., Martin, P. A., Mark, Shavit, N. and Guy L. Steele, J., 2004. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. *Proceedings of the 16th annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*.

El-Gahazawi and Chauvin, S., 2001. UPC Benchmarking Issues. *IEEE Proceedings of the International Articles on Parallel Processing (ICPP'01)*.

Foster, I., 2001. Parallel Computing in 2010: opportunities and challenges in a networked world. *ACM SIGPLAN Notices*, **36**, 1.

Goel, A., Roychoudhury, A. and Mitra, T., 2003. Compactly Representing Parallel Program Executions. *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*.

Grama, A., Gupta, A., Karypis, G. and Kumar, V., 2003. Introduction to Parallel Computing. Addison Wesley, second edition.

Gray, R., Kotz, D., Nog, S., Rus, D. and Cybenko, G., 1997. Mobile Agents: The Next Generation in Distributed Computing. *IEEE 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis (pAs'97)*, pages 8–24.

Grundmann, T., Ritt, M. and Rosenstiel, W., 2000. TPO++: An object-oriented message-passing library in C++. *IEEE Proceedings of the International Articles on Parallel Processing (ICPP'00)*.

Guitart, J., Torres, J., Ayguade, E. and Bull, J. M., 2001. Performance Analysis Tools For Parallel Java Applications on Shared-memory Systems. *IEEE Proceedings of the 2001 International Article on Parallel Processing (ICPP)*.

Hamdan, M. M., 2000. A Combinational Framework for Parallel Programming using Algorithmic Skeletons. *PhD thesis*, Heriot-Watt University.

Hammond, K. and Michaelson, G., 1999. Research Directions in Parallel functional Programming. Springer.

Herlihy, M., 1991. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems*, **11**, 124–149.

Herrmann, C. and Lengauer, C., 2000. HDC: A Higher Order Language for Divide and Conquer. *Parallel Processing Letters*, **10**, 239–250.

Hicks, J., Chiou, D., Ang, B. S. and Arvind, 1993. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, **18**, 273–300.

Hill, J. M. D., McColl, B., Stefanescu, D. C., Goudreau, M. W., Lang, K., Rao, S. B., Suel, T., Tsantilas, T. and Bisseling, R., 1997. BSPlib: The BSP Programming Library. *Teknik Rapor*, Oxford University Computing Laboratory.

HPFForum. http://dacnet.rice.edu/Depts/CRPC/HPFF/index.cfm.

IBMTspaces. http://www.almaden.ibm.com/cs/TSpaces/html/UserGuide.html.

JavaThreads. http://java.sun.com/docs/books/tutorial/essential/threads/.

Jones, S. L. P. and A. Gordon, S. F., 1996. Concurrent Haskell. *23rd ACM Symposium on Principles of Programming Languages*, **1**, 295–308.

Kale, L. V. and Krishnan, S., 1993. Charm++: a portable concurrent object oriented system based on C++. *Proceedings of the 8th Annual Article on Object Oriented Programming Systems, Languages and Applications*.

Kuchen, H., 2002. A Skeleton Library. *Euro-Par 2002, LNCS 2400*, **1**, 620–629.

Kuchera, W. and Wallace., C., 2004. The UPC Memory Model: Problems and Prospects. *IEEE Proceedings of the 18th International Parallel and Distributed Processing Syposium (IPDPS'04)*.

Lane, T. G., 1995. Recent Enhancements to PVM. *The International Journal of Supercomputer Applications and High Performance Computing*, **9**.

Larus, J., 1999. Whole program paths. *Proceedings of the ACM SIGPLAN Article on Programming Language Design and Implementation (PLDI'03)*.

Leopold, C., 2001. Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches. John Wiley and Sons.

Loidl, H. W., F. Rubio, N. S., Hammond, K., Horiguchi, S., Kluisik, U., Loogen, R., Michaelson, G., Pena, R., Priebe, S., Rebon, A. J. and Trinder, P. W., 2003. Comparing Parallel Functional Languages: Programming and Performance. *Kluwer Academic Publishers Higher-Order and Symbolic Computation*, **16**, 203–251.

Loidl, H. W., Klusik, U., Hammond, K., Loogen, R. and Trinder, P., 2000. GpH and Eden: Comparing Two Parallel Functional Languages on a Beowful Cluster. *Scottish Functional Programming Workshop (SFP'00), rends in Functional Programming*, **2**, 39–52.

Loogen, R., 1999. Programming Language Constructs in (Research Directions in Parallel functional Programming). Springer.

MacDonald, S., Anvik, J., Bromling, S., Schaeffer, J., Szafron, D. and Tan, K., 2002. From patterns to frameworks to parallel programs. *Parallel Computing*, **28**, 1663–1668.

MESSENGERS. http://www.ics.uci.edu/ bic/messengers.

MobileAgents. http://di002.edv.uniovi.es/ arturop/MobAgentsEng.html.

MPIForum. http://www.mpi-forum.org.

Nikhil, R. S. and Arvind, 2001. Implicit Parallel Programming in pH. Morgan Kaufmann.

O'Callahan, R. and Choi, J.-D., 2003. Hybrid dynamic data race detection. *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*.

OpenMP. http://www.openmp.org/drupal.

Ortega, Y. and Pena, R., 1998. Looking for Eden in the land of parallel-functional languages. *Workshop on Parallel Functional Programming in association with IFL'98*.

Parallaxis. http://www.ee.uwa.edu.au/ braunl/parallaxis/.

POSIXThreads. http://moss.csc.ncsu.edu/ mueller/pthreads/.

Pozniansky, E. and Schuster, A., 2003. Efficient On-the-Fly Data Race Detection in Multithreaded C++ programs. *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*.

Problem Solving Environments Home Page. http://www-cgi.cs.purdue.edu/cgi-bin/acc/pses.cgi.

Rabhi, F. A. and Gorlatch, S., 2002. Patterns and Skeletons for Parallel and Distributed Computing. Springer.

Roy, P. V., Brand, P., Duchier, D., Haridi, S., Henz, M. and Schulte, C., 2003. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, **3**, 717–763.

Saunders, S. and Rauchwerger, L., 2003. ARMI: An Adaptive, Platform Independent Communication Library. *ACM Principles and Practice of Parallel Programming (PPoPP'03)*.

Scaife, N., Michaelson, G. and Horiguchi, S. Parallel Standard ML with Skeletons.

Sérot, J. and Ginhac, D., 2002. Skeletons for parallel image processing: an overview of the SKiPPER project. *Parallel Computing*, **28**, 1785–1808.

Serrarens, P., 1998. More or Less Explicit Parallelism in Concurrent Clean. *Workshop on Parallel Functional Programming in association with IFL'98*.

Skillicorn, D. B. and Talia, D., 1998. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, **30**(2), 123–169.

Suzuki, N. and Fukuda, M., 1999. Self-Migrating Threads for Multi-Agent Applications. *IEEE Computer Society International Workshop on Cluster Computing*.

Taylor, F. S., 1993. Parallel Functional Programming by Partitioning. *PhD thesis*, University of London, London, UK.

WAVEGroup. http://www-zorn.ira.uka.de/wave/wave.html.

Wikipedia. http://en.wikipedia.org/wiki/Classification.

Windsor, A., 2004. An NC algorithm for finding a maximal acyclic set in a graph. *Proceedings of the 16th annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*, pages 145 – 150.

Yang, J., 1997. Coordination Based Structured Parallel Programming. *PhD thesis*, University of London.

A P P E N D I X

A. Time Measurement for Matrix - Vector Multiply

A.1. OpenMP, SMP

Sun Fire E6900 (less processors, but SMP machine).

```
SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix1

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
| on host sunc10.rz.RWTH-Aachen.DE
| SGE Request: 970667   Queue: sunc10.q
------------------------------------------------

12.949615
12.530919
13.734507
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:00:31 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:54
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix2

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
| on host sunc00.rz.RWTH-Aachen.DE
| SGE Request: 970668   Queue: sunc00.t
------------------------------------------------

6.510820
6.477644
5.747368
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:00:07 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:30
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix4

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
```

```
| on host sunc13.rz.RWTH-Aachen.DE
| SGE Request: 970669   Queue: sunc13.q
------------------------------------------------

3.749756
3.995276
3.491776
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:00:22 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:45
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix8

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
| on host sunc00.rz.RWTH-Aachen.DE
| SGE Request: 970670   Queue: sunc00.t
------------------------------------------------

1.687861
1.625732
1.676302
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:59:54 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:17
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix16

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
| on host sunc00.rz.RWTH-Aachen.DE
| SGE Request: 970671   Queue: sunc00.t
------------------------------------------------

0.886834
0.896128
0.877115
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:59:51 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:15
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix32
```

```
------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:15:30 MET 2004
| on host sunc00.rz.RWTH-Aachen.DE
| SGE Request: 970698   Queue: sunc00.t
------------------------------------------------

0.497120
0.494220
0.491827
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:16:39 MET 2004
| peak memory value: N/A
| real time used: 00:01:09
------------------------------------------------
```

A.2. OpenMP, ccNUMA

```
more outOmpMatrix1g
------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:30:30 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970631   Queue: sunc16.q
------------------------------------------------

14.554998
13.915468
14.061663
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:32:42 MET 2004
| peak memory value: N/A
| real time used: 00:02:13
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix2g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:41:46 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970636   Queue: sunc16.q
------------------------------------------------

10.334152
8.363506
10.364811
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:43:45 MET 2004
```

```
| peak memory value: 1.68G
| real time used: 00:02:00
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix4g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:47:22 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970637   Queue: sunc16.q
-------------------------------------------------

5.118155
5.254976
5.532442
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:49:12 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:50
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix8g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:47:22 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970638   Queue: sunc16.q
-------------------------------------------------

3.748879
2.785684
2.886551
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:49:07 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:45
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix16g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:15:30 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970639   Queue: sunc16.q
-------------------------------------------------

2.498763
1.764210
```

```
2.291116
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:17:06 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:36
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix32g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:15:46 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970640   Queue: sunc19.q
-------------------------------------------------

2.400865
2.436438
2.375442
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:17:27 MET 2004
| peak memory value: 1.68G
| real time used: 00:01:41
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/matrix[!]$ more outOmpMatrix64g
more outOmpMatrix64g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:10:08 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970643   Queue: sunc19.q
-------------------------------------------------

2.926020
3.050064
2.963061
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:11:48 MET 2004
| peak memory value: 1.69G
| real time used: 00:01:40
-------------------------------------------------
SUNOS:sunc20:/work/aa006le/OpenMP2/matrix[!]$ more outOmpMatrix128g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 21:49:59 MET 2004
| on host sunc17.rz.RWTH-Aachen.DE
| SGE Request: 970644   Queue: sunc17.q
-------------------------------------------------
```

```
5.246612
4.221971
4.267757
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 21:51:38 MET 2004
| peak memory value: N/A
| real time used: 00:01:39
-------------------------------------------------
```

A.3. MPI, ccNUMA

```
time mprun -np 1 prog

real    0m26.840s
user    0m0.020s
sys     0m0.036s
SUNOS:sunc20:/work/aa006su/MPI2/matrix[!]$ time mprun -np 2 prog
34.819734,
real    0m35.958s
user    0m0.018s
sys     0m0.038s

mprun -np 2 prog
35.070022,

more outMpiMatrix4g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:21:07 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 970963   Queue: sunc18.p
-------------------------------------------------

44.224915,
44.211996,
44.474745,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:23:57 MET 2004
| peak memory value: 15.85M
| real time used: 00:02:50
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/matrix[!]$ more outMpiMatrix8g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:26:42 MET 2004
```

```
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970964   Queue: sunc19.p
------------------------------------------------

47.976364,
47.928871,
47.300173,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:30:07 MET 2004
| peak memory value: 15.85M
| real time used: 00:03:25
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/matrix[!]$ more outMpiMatrix16g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:21:16 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970965   Queue: sunc16.p
------------------------------------------------

47.960360,
49.399255,
50.303306,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:24:53 MET 2004
| peak memory value: 17.55M
| real time used: 00:03:38
------------------------------------------------------------
```

A.4. MPI, Beowulf

```
  np 1      np 2     np 4      np 8     np 16
5.895872,7.880361,14.623500,12.665565,19.528374
4.574467,7.650136,11.100957,13.241552,19.311171
5.625762,7.823462,10.938837,11.564033,17.162392
7.824083,9.071199,9.921196,11.427652,14.711691
5.229454,7.741427,10.111857,11.240894,14.113169
6.731793,9.306816,10.939820,11.374350,13.183217
```

B. Time Measurement for Mergesort

B.1. OpenMP, ccNUMA

B.1.1. Best Case

```
more outOmpMergeBest1g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:30:30 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970630   Queue: sunc16.q
-------------------------------------------------

80.127588,
80.666249,
82.985441,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:34:47 MET 2004
| peak memory value: 768.09M
| real time used: 00:04:17
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/best[!]$ more outOmpMergeBest2g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:53:17 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970646   Queue: sunc16.q
-------------------------------------------------

50.082234,
48.944336,
47.537940,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:55:58 MET 2004
| peak memory value: 1.12G
| real time used: 00:02:41
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/best[!]$ more outOmpMergeBest4g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:53:17 MET 2004
```

```
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970647   Queue: sunc16.q
------------------------------------------------

32.837522,
33.257889,
32.521403,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:55:12 MET 2004
| peak memory value: 1.50G
| real time used: 00:01:55
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/best[!]$ more outOmpMergeBest8g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:58:37 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970648   Queue: sunc16.q
------------------------------------------------

23.751311,
23.671898,
23.658460,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:00:06 MET 2004
| peak memory value: N/A
| real time used: 00:01:29
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/best[!]$

---------------

time ./prog 1
82.582161,
real    1m24.894s
user    1m10.829s
sys     0m1.169s
SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ time ./prog 2
41.856210,
real    0m44.982s
user    1m14.043s
sys     0m2.576s
SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ ./prog 4
29.807576,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ ./prog 8
21.287086,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ ./prog 16
22.129667,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ ./prog 32
```

```
20.819331,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/best[!]$ ./prog 48
```

B.1.2. Random Case

```
------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:53:16 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970659   Queue: sunc16.q
------------------------------------------------

only one thread does sorting
61.997367,
only one thread does sorting
63.070206,
only one thread does sorting
60.379645,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:56:51 MET 2004
| peak memory value: 768.10M
| real time used: 00:03:35
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/random[!]$ more outOmpMergeRand2g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:04:14 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970674   Queue: sunc16.q
------------------------------------------------

45.002183,
39.925784,
47.287158,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:07:06 MET 2004
| peak memory value: 1.12G
| real time used: 00:02:52
------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/random[!]$ more outOmpMergeRand4g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:04:15 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970675   Queue: sunc16.q
------------------------------------------------

30.667217,
33.434800,
```

```
31.570635,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:06:26 MET 2004
| peak memory value: 1.50G
| real time used: 00:02:11
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/random[!]$ more outOmpMergeRand8g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:09:53 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970676   Queue: sunc16.q
-------------------------------------------------

25.173901,
23.651243,
23.407918,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:11:38 MET 2004
| peak memory value: 1.87G
| real time used: 00:01:45
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/random[!]$ more outOmpMergeRand16g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:21:11 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970677   Queue: sunc16.q
-------------------------------------------------

unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
SUNOS:sunc20:/work/aa006su/OpenMP2/merge/random[!]$

------------

SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 1
56.407192,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 2
44.769378,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 1
59.801905,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 4
30.626349,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 8
36.365037,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 16
24.652945,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 32
```

```
35.307240,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/random[!]$ ./prog 48
```

B.1.3. Worst Case

```
more outOmpMergeWorst1g

--------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 08:53:17 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970663   Queue: sunc16.q
--------------------------------------------------

only one thread does sorting
81.862861,
only one thread does sorting
79.980057,
only one thread does sorting
79.749710,
--------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 08:57:30 MET 2004
| peak memory value: 768.10M
| real time used: 00:04:14
--------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/worst[!]$ more outOmpMergeWorst2g

--------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:04:15 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970681   Queue: sunc16.q
--------------------------------------------------

53.969241,
51.492970,
45.149602,
--------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:07:06 MET 2004
| peak memory value: 1.12G
| real time used: 00:02:51
--------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/worst[!]$ more outOmpMergeWorst4g

--------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:04:15 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 970682   Queue: sunc16.q
--------------------------------------------------
```

```
32.685181,
30.685599,
32.434474,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:06:07 MET 2004
| peak memory value: 1.50G
| real time used: 00:01:52
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/worst[!]$ more outOmpMergeWorst8g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:27:09 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970683   Queue: sunc19.q
-------------------------------------------------

27.907675,
25.288126,
23.920904,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 09:28:46 MET 2004
| peak memory value: 1.87G
| real time used: 00:01:37
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/OpenMP2/merge/worst[!]$ more outOmpMergeWorst16g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 09:21:17 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970684   Queue: sunc19.q
-------------------------------------------------

unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
unknown signal type = 11 (stage 2)
SUNOS:sunc20:/work/aa006su/OpenMP2/merge/worst[!]$

---------------

SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 1
70.385525,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 2
56.251616,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 4
56.238972,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 8
26.062037,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 16
```

```
25.562178,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 32
35.100334,SUNOS:sunc20:/work/aa006le/OpenMP2/merge/worst[!]$ ./prog 48
30.206300,
```

B.2. MPI, ccNUMA

B.2.1. Best Case

```
Some measurements are interactive as for strange reasons these jobs were
killed in the batch system:

mprun -np 1 prog
66.125804,SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ mprun -np 4 prog
28.272233,SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ mprun -np 2 prog
42.625526,
mprun -np 8 prog
21.363042,
mprun -np 16 prog
18.151412,SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ mprun -np 32 prog
mprun: Not enough node cpus are available to satisfy resource request. (Try
reducing the -np value. If not using -x, try using -S or -W.)

more outMpiBest1ga

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 17:20:09 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 971797   Queue: sunc16.p
-------------------------------------------------

90.023958,
error: executing task of job 971797 failed:
[Job sge.971797 on sunc16.rz.RWTH-Aachen.DE: 1 of 1 processes did not start]
Job sge.971797 on sunc16.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

78.994747,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 17:23:46 MET 2004
| peak memory value: 11.80M
| real time used: 00:03:38
-------------------------------------------------

more outMpiBest2g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 11:02:27 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970848   Queue: sunc19.p
```

```
------------------------------------------------

54.046144,
54.811959,
53.597686,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 11:06:21 MET 2004
| peak memory value: 15.85M
| real time used: 00:03:54
------------------------------------------------

more outMpiBest4ga

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 19:29:26 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 971825   Queue: sunc16.p
------------------------------------------------

38.364021,
error: executing task of job 971825 failed:
error: executing task of job 971825 failed:
[Job sge.971825 on sunc18.rz.RWTH-Aachen.DE: 1 of 1 processes did not start]
Job sge.971825 on sunc19.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

37.044796,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 19:31:57 MET 2004
| peak memory value: 14.68M
| real time used: 00:02:31
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ more outMpiBest8g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:15:41 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970920   Queue: sunc19.p
------------------------------------------------

27.499330,
28.540497,
31.730463,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:18:02 MET 2004
| peak memory value: 15.85M
| real time used: 00:02:24
```

```
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ more outMpiBest16g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:04:16 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 970922   Queue: sunc18.p
------------------------------------------------

24.501523,
22.788031,
23.394890,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:06:25 MET 2004
| peak memory value: 14.60M
| real time used: 00:02:09
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ more outMpiBest32g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 11:30:59 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 970925   Queue: sunc18.p
------------------------------------------------

24.317219,
33.954209,
31.622065,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 11:33:58 MET 2004
| peak memory value: 15.84M
| real time used: 00:02:59
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$ more outMpiBest64g
outMpiBest64g: No such file or directory
SUNOS:sunc20:/work/aa006su/MPI2/merge/best[!]$
```

B.2.2. Random Case

```
mprun -np 1 prog
49.968823,SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ mprun -np 2 prog
34.243002,SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ mprun -np 4 prog
25.281914,SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$
SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ mprun -np 8 prog
19.395883,SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ mprun -np 16 prog
16.533749,
```

```
more outMpiRand1g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:09:57 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 971001   Queue: sunc19.p
-------------------------------------------------

60.166358,
error: executing task of job 971001 failed:
[Job sge.971001 on sunc19.rz.RWTH-Aachen.DE: 1 of 1 processes did not start]
Job sge.971001 on sunc19.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

59.735608,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:13:32 MET 2004
| peak memory value: 11.80M
| real time used: 00:03:35
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand2g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 13:40:16 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 971229   Queue: sunc19.p
-------------------------------------------------

[Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: 2 of 2 processes did not start]
Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

[Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: 2 of 2 processes did not start]
Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

[Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: 2 of 2 processes did not start]
Job sge.971229 on sunc19.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 13:42:06 MET 2004
| peak memory value: 10.10M
| real time used: 00:01:50
-------------------------------------------------
```

```
more outMpiRand2ga

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 17:31:23 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 971822   Queue: sunc19.p
------------------------------------------------

55.652203,
45.895615,
43.548044,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 17:34:58 MET 2004
| peak memory value: 15.85M
| real time used: 00:03:35
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand4g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:26:41 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 971002   Queue: sunc18.p
------------------------------------------------

31.065660,
31.978393,
32.337318,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:29:12 MET 2004
| peak memory value: 15.85M
| real time used: 00:02:31
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand8g

------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:37:54 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 971003   Queue: sunc16.p
------------------------------------------------

24.319575,
24.709012,
26.715976,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:40:11 MET 2004
```

```
| peak memory value: 17.55M
| real time used: 00:02:18
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand16g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:32:23 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 971004   Queue: sunc16.p
-------------------------------------------------

24.481873,
22.326215,
23.104005,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:34:49 MET 2004
| peak memory value: 17.55M
| real time used: 00:02:26
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand32g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 11:53:06 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 971005   Queue: sunc18.p
-------------------------------------------------

26.254641,
25.728889,
27.400871,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 11:55:51 MET 2004
| peak memory value: 15.84M
| real time used: 00:02:45
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$ more outMpiRand64g
outMpiRand64g: No such file or directory
SUNOS:sunc20:/work/aa006su/MPI2/merge/random[!]$
```

B.2.3. Worst Case

```
mprun -np 1 prog
66.149647,SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ mprun -np 2 prog
41.119770,SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ mprun -np 4 prog
27.627378,SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ mprun -np 8 prog
20.312268,SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ mprun -np 16 prog
```

```
17.283030,

more outMpiW1ga

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 17:36:56 MET 2004
| on host sunc16.rz.RWTH-Aachen.DE
| SGE Request: 971824   Queue: sunc16.p
-------------------------------------------------

84.222697,
error: executing task of job 971824 failed:
[Job sge.971824 on sunc16.rz.RWTH-Aachen.DE: 1 of 1 processes did not start]
Job sge.971824 on sunc16.rz.RWTH-Aachen.DE: aborted due to an unexpected
error.

92.984256,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 17:40:47 MET 2004
| peak memory value: 11.80M
| real time used: 00:03:51
-------------------------------------------------

more outMpiW2g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 11:13:46 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970911   Queue: sunc19.p
-------------------------------------------------

55.694205,
50.352307,
51.173915,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 11:17:15 MET 2004
| peak memory value: 15.85M
| real time used: 00:03:29
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ more outMpiW4g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:09:54 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970950   Queue: sunc19.p
-------------------------------------------------
```

```
34.062280,
32.869238,
33.430165,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:12:23 MET 2004
| peak memory value: 15.85M
| real time used: 00:02:29
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ more outMpiW8g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:21:08 MET 2004
| on host sunc19.rz.RWTH-Aachen.DE
| SGE Request: 970951   Queue: sunc19.p
-------------------------------------------------

28.080131,
27.183745,
32.116552,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:23:52 MET 2004
| peak memory value: 15.85M
| real time used: 00:02:44
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ more outMpiW16g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 12:15:28 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
| SGE Request: 970952   Queue: sunc18.p
-------------------------------------------------

21.725114,
22.179005,
22.544710,
-------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 12:17:40 MET 2004
| peak memory value: 15.83M
| real time used: 00:02:12
-------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ more outMpiW32g

-------------------------------------------------
| Execution of Batch-Request started at Fri Dec 17 11:42:14 MET 2004
| on host sunc18.rz.RWTH-Aachen.DE
```

```
| SGE Request: 970953   Queue: sunc18.p
------------------------------------------------

22.343896,
24.109160,
28.509685,
------------------------------------------------
| Execution of Batch-Request stopped at Fri Dec 17 11:44:40 MET 2004
| peak memory value: 15.84M
| real time used: 00:02:27
------------------------------------------------

SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$ more outMpiW64g
outMpiW64g: No such file or directory
SUNOS:sunc20:/work/aa006su/MPI2/merge/worst[!]$
```

B.3. MPI, Beowulf, Small N

N = 1.000.000

B.3.1. Best Case

```
  np 1    np 2    np 4    np 8    np 16   np 32   np 64
1.57000,1.76000,1.97000,1.45000,1.30000,1.55000,1.480000,
1.25000,1.56000,2.42000,2.42000,1.76000,1.53000,1.490000,
1.67000,1.79000,1.78000,2.01000,1.33000,1.51000,1.500000,
1.77000,1.73000,1.77000,1.59000,1.52000,1.64000,2.130000,
2.34000,1.97000,2.04000,1.62000,1.51000,1.54000,1.480000,
1.56000,1.53000,1.55000,1.41000,1.49000,1.58000,1.590000,
1.81000,1.73000,2.04000,1.68000,1.68000,2.14000,1.870000,
1.37000,1.53000,1.49000,1.50000,1.58000,1.44000,2.110000,
1.81000,2.35000,1.91000,1.61000,1.66000,1.58000,1.440000,
2.02000,1.92000,1.89000,1.69000,1.68000,1.51000,1.520000,
1.58000,2.58000,2.39000,1.87000,1.56000,1.57000,1.460000,
1.94000,1.88000,2.07000,1.68000,1.51000,1.49000,1.430000,
1.42000,1.87000,2.02000,1.48000,1.55000,1.43000,1.400000,
1.80000,1.58000,2.90000,2.23000,1.86000,1.56000,1.490000,
```

B.3.2. Random Case

```
  np 1    np 2    np 4    np 8    np 16   np 32   np 64
3.35000,2.57000,2.12000,1.93000,1.34000,1.55000,1.49000,
3.04000,2.76000,2.13000,2.69000,2.26000,2.07000,1.62000,
3.45000,2.64000,2.36000,1.87000,1.33000,1.64000,1.48000,
2.69000,2.12000,2.25000,1.48000,1.29000,1.43000,1.83000,
3.41000,2.76000,3.09000,2.25000,1.74000,1.63000,1.61000,
3.56000,2.78000,2.13000,1.88000,1.67000,1.60000,1.65000,
3.03000,2.62000,1.97000,1.81000,1.58000,1.85000,2.10000,
3.63000,2.10000,2.21000,1.95000,1.56000,1.79000,2.18000,
```

```
3.83000,3.53000,2.34000,1.85000,1.65000,1.37000,1.53000,
3.50000,2.44000,2.24000,1.91000,1.62000,1.58000,1.41000,
2.70000,2.48000,2.11000,2.48000,2.02000,2.01000,1.55000,
3.79000,2.92000,1.90000,1.89000,1.43000,1.56000,1.41000,
2.64000,2.19000,2.01000,1.61000,1.37000,1.55000,1.38000,
2.89000,2.59000,2.34000,2.53000,1.80000,1.69000,1.81000,
```

B.3.3. Worst Case

```
  np 1    np 2    np 4    np 8    np 16   np 32   np 64
2.30000,2.98000,2.63000,2.37000,1.68000,1.73000,1.70000,
2.33000,2.51000,2.17000,1.75000,1.63000,1.48000,1.43000,
2.68000,2.48000,1.68000,1.69000,1.62000,2.23000,1.79000,
3.28000,2.45000,1.84000,1.91000,1.60000,1.58000,1.50000,
2.91000,2.32000,1.83000,1.60000,1.48000,1.50000,1.51000,
2.87000,2.49000,1.97000,1.61000,2.25000,2.03000,1.73000,
3.11000,2.10000,2.06000,1.94000,1.58000,1.56000,1.47000,
3.39000,2.73000,2.19000,1.78000,1.73000,1.48000,1.54000,
2.89000,2.49000,2.10000,1.88000,1.61000,1.44000,1.30000,
2.89000,2.68000,2.72000,2.30000,1.70000,1.55000,1.65000,
2.87000,1.92000,1.86000,1.55000,1.53000,1.55000,1.45000,
2.83000,1.82000,1.56000,1.60000,1.61000,1.55000,2.42000,
2.41000,2.95000,2.86000,2.15000,1.66000,1.57000,1.57000,
2.48000,2.17000,2.15000,1.85000,1.63000,1.60000,1.49000,
```

B.4. MPI, Beowulf, Large N

N = 100.000.000

B.4.1. Best Case

```
  np 1    np 2    np 4    np 8    np 16   np 32
29.792241,42.320223,41.994550,38.204744,43.696682
37.113114,36.508190,41.682661,43.283328,43.431303
37.036981,35.538042,42.841430,44.575356,43.941911
34.984744,35.873920,42.923187,43.069935,38.245744
```

B.4.2. Random Case

```
75.852566,59.328657,46.049081,42.064306,45.399370,p10_4829:
p4_error: net_recv read:  probable EOF on socket: 1
p3_9312:  p4_error: net_recv read:  probable EOF on socket: 1
p18_28809:  p4_error: net_recv read:  probable EOF on socket: 1
p1_14734:  p4_error: net_recv read:  probable EOF on socket: 1
p22_27451:  p4_error: net_recv read:  probable EOF on socket: 1
p2_32178:  p4_error: net_recv read:  probable EOF on socket: 1
p24_31433:  p4_error: net_recv read:  probable EOF on socket: 1
p11_14952:  p4_error: net_recv read:  probable EOF on socket: 1
p4_29206:  p4_error: net_recv read:  probable EOF on socket: 1
p9_10378:  p4_error: net_recv read:  probable EOF on socket: 1
```

```
p19_1707:  p4_error: net_recv read:  probable EOF on socket: 1
p6_13387:  p4_error: net_recv read:  probable EOF on socket: 1
p14_25040:  p4_error: net_recv read:  probable EOF on socket: 1
p13_17984:  p4_error: net_recv read:  probable EOF on socket: 1
p5_7323:  p4_error: net_recv read:  probable EOF on socket: 1
p12_3734:  p4_error: net_recv read:  probable EOF on socket: 1
```

B.4.3. Worst Case

```
  np 1    np 2    np 4    np 8    np 16   np 32
56.425412,51.296409,47.800027,41.404835,40.701334
57.973493,49.273044,43.528942,46.059671,47.044458
58.663148,48.198275,45.218142,46.371781,44.989813
53.457417,51.021359,48.293039,46.620663,43.733794
```

BIOGRAPHY

Born on September 22, 1978 in İstanbul, Turkey, Ayşe Beliz Şenyüz started her education in Şişli Terakki Primary School and continued in Notre Dame de Sion French High School from where she was graduated in June 1997. Between 1997 and 2002 she studied computer engineering in Galatasaray University, İstanbul. In September 2002 she was accepted to Computer Engineering M. Sc. Program at İstanbul Technical University and at the same time she started working as a research assistant at Galatasaray University. During that period, she worked on computer networks and parallel programming under the supervision of Prof. Dr. A. Emre Harmancı.