Tesis de Maestría en Informática
Programa de Desarrollo de las Ciencias Básicas
Montevideo - Uruguay

Modeling PC-based Clusters for Parallel Computing.

Ariel Sabiguero Yawelak

2002

(mailto:asabigue@fing.edu.uy)
Centro de Cálculo - Departamento de Investigación Operativa
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
República Oriental del Uruguay

Abstract

Parallelism is a Computer Science discipline that is gaining a more important place in data centers. Since mid-nineties parallel computers built out of commodity components and free software started to show up and introduced new challenges to computer industry. Before that time, parallelism and supercomputing were almost constraint to multi-million projects on aerospace, military simulations, meteorology, etc. Since Beowulf experiences and GNU/Linux success as a solid OS, the paradigms started to change and several government and private initiatives demonstrated that this kind of parallel system can be exploited. Several new applications are arising also. We understood that it was important to develop mathematical basis that help in the task of building this kind of parallel systems and assist the hardware architect to devise an optimal parallel machine. This work presents a set of model templates that can be instantiated so as to model and predict performance estimators for given applications running on specific hardware. The space of parallel applications is partitioned using an adequate taxonomy and a model template is built for each class of the taxonomy. The mathematical abstraction used for the models are the Stochastic Petri Networks. Two real application examples are analyzed in depth so the model usage is shown.

El paralelismo es una disciplina Informática que está ganando un lugar más importante en los centros de cómputos. Desde mediados de los noventa aparecieron computadoras paralelas construidas con componentes estándar de bajo costo y software libre que presentaron nuevos desafíos a la industria de la computación. Antes de ese entonces, el paralelismo y el procesamiento en supercomputadoras prácticamente estaba restringido a proyectos multimillonarios de la industria aeroespacial, simulaciones militares, meteorología, etc. Desde las experiencias de Beowulf y el éxito de GNU/Linux como un sistema operativo sólido, los paradigmas comenzaron a cambiar y múltiples iniciativas privadas y gubernamentales demostraron que este tipo de sistemas paralelos pueden ser aprovechados. Nuevas aplicaciones se están creando para estos sistemas. Consideramos que era importante desarrollar bases matemáticas que ayuden en la tarea de construir este tipo de sistemas paralelos y asistir al arquitecto de hardware para crear una máquina paralela óptima. Este trabajo presenta una serie de plantillas de modelos que pueden ser instanciados de forma de modelar y predecir estimadores de desempeño para aplicaciones específicas ejecutadas en ciertas máquinas. El espacio de aplicaciones paralelas es particionado de acuerdo a una taxonomía y una plantilla de modelo se construye para cada clase de la taxonomía. La abstracción matemática utilizada para los modelos son las Redes de Petri Estocásticas. Dos ejemplos de aplicación se analizan en profundidad para mostrar el uso de los modelos.

Keywords

Parallel performance evaluation, cluster computing, parallel programming, Stochastic Petri Networks, parallel programming taxonomy,

Evaluación de desempeño paralelo, computación en grupos, programación paralela, Redes de Petri Estocásticas, taxonomías de programas paralelos.

i. Contents

ii. Preface

This document is a master degree thesis on the field of parallelism. It is a partial result within an initiative of commodity parallelism studies started back on 1997 at the CeCal (Centro de Cálculo – Numerical Analysis Department – Facutlad de Ingeniería – Universidad de la República) when a tanker spilled millions of liters of petroleum near Punta del Este, an important seaside resort in our country. CeCal already modeled the flows of the Río de la Plata and had a cluster (FDDI/Power/AIX) with four nodes and 7 CPUs that could help with the modeling of the pollutant dispersion. With that accident and parallel experiences in mind a group was set up under the name ParEnO to study the possibilities of parallelism over standard and available everywhere computing components. The group initially existed within the CeCal but later the CeCal and the InCo (Instituto de Computación – Computer Science Institute – Facutlad de Ingeniería – Universidad de la República) were merged within the Engineering Faculty into the InCo and different people moved to different groups. People working with algorithms joined the Programming groups while people dealing with modeling and simulation moved to the Optimization group.
At that time the availability of Pentium based PCs was growing and the raw numerical performance of such processors seemed enough for number crunching. Also the first papers of Beowulf results started shocking the parallel community. It started to become clear that mass market forces were able to produce standalone systems that could be compared with high end ones. On the other hand, networking technologies were also making high speed switched standard networks available. Also initiatives from the FSF, GNU and the Linux community turned into complete free software solutions that solves not only the OS but compilers, libraries, development environments, schedulers, etc.
The ParEnO group realized that there was a potential parallel cluster almost in every office and different study branches started from there on. Some study groups analyzed the possibilities of porting parallel software to the available OSs (most generally Windows), others worked on the idea of multiple OS booting systems that can flip from commercial functions during office hours to parallel clusters during the night.
Along with this different branches grew the need for modeling these parallel systems. Researching this area, we found that several authors have already modeled the performance of groups of systems addressing parallel problems but from specific aspects, like those on [ABC1], [BUY1], [BRO1], [LIN1] and others. There exists excellent models that deal with the impact of network aspects (speed, latency, reliability, etc.) on the overall performance of a system, or maybe the impact of multiple processors sharing resources. Also other approaches to performance were done from the point of view of the parallel algorithms and comparisons to classic uniprocessor implementations. We found that there was no modeling of both the system and the algorithm implementation as a whole. As we were evaluating parallelism as a tool to be applied to particular projects, we understood that there was a need for such kind of system performance modeling. We believe that it is important to model performance descriptors that allows a company to build a cluster that helps solving some problem and decide if it is better to invest in few high-power CPUs or multiple low cost ones or invest in network hardware, etc.
Funding for this line of research came from the Clemente Estable under project number 4072 named "***Modelado y construcción de una máquina paralela virtual con componentes de bajo costo (1999-2001)***" (*Modeling and construction of a parallel virtual machine with low cost components*). The project built a tiny cluster for theoretical performance research, education on several grade curses and grade thesis at the University and was used for several

performance tests.
System performance evaluation or algorithm performance evaluation are beyond the scope of this study. Joint system and algorithm performance evaluation are considered the target of this work. The present thesis work goal is to obtain an overall system performance modeling that can be used to predict particular algorithm execution times and also to assist in the construction of clusters.

iii. Assumed Knowledge

Readers would be required to have basic knowledge of computer and processor architecture, computer networks, Petri networks, parallelism and parallel tools. Background in different operating systems is also advisable.

iv. Organization

1: Introduction: Here we present the background, motivations and scope of current research. It presents some commodity computing projects and introduces motivations for continuing the research and development of such technologies.

2: Taxonomy and Parallelism: This chapter introduces many of the concepts that are used in the rest of the work. It defines different parallel computing scenarios and determines the specific scope of parallel problems addressed. The parallel problem taxonomy, mathematical abstraction used and simulation tools chosen for analysis are also introduced here. Knowledge of basic and stochastic Petri networks is assumed. For the reader not familiar with Petri networks, we suggest the reading of [SAB1].

3: Modeling distributed.net's RC5: We selected a popular worldwide commodity parallel initiative to discuss some relevant aspects that will prove useful when we present later the detailed analysis of each class of parallel applications. This chapter fundaments some technical aspects of our analysis and discusses specific modeling scale details. This problem is useful to help separating relevant performance aspects from non relevant ones. It shows why it is necessary to collapse multiple very small details into single performance descriptors when whole application execution is aimed at.

4: Model templates for general parallel applications: This chapter is very ambitious and is the core of our research. It applies systematically all aspects of our study to all classes of parallel applications introduced in the second chapter and explains how to ascertain two specific estimations of performance: Total Execution Time (TET) and Mean Execution Speed (MES). This chapter presents methods for constructing Petri nets that models each specific problem on specific hardware configurations out of model templates. It also explains how to determine relevant performance figures and shows how to compute our performance estimators.

5: Case studies: Here we present two case studies of two different problems of different parallel classes: *task farming* and *SPMD*. In both studies we follow the procedure suggested in the third chapter strictly so as to determine the performance estimation parameters TET and MES. Single processor execution was performed to obtain required individual performance parameters of the system. Parallel execution was both simulated and benchmarked and numerical results were compared and discussed.

6: Conclusions and future work: Here we summary the work, analyze application environments for the developed template models and present related future work and projects.

v. Acknowledgments

This is a difficult paragraph in which one is supposed to recall all the people that helped in this work, which is not simple as it spanned over so many months. Many people always help indirectly and is not mentioned. If you are reading this and feel that helped me, be sure I feel so too, and that I am infinitely grateful.
But my week memory is able to keep some people in mind, here I go. First I would like to thank Héctor Cancela, my patient tutor that helped me so much all way long. I would also like to say thanks to all the people at the Investigación Operativa department, the former CeCal and the In.Co. that directly or indirectly helped also. Thank you people, it is a great honor to work with you all.
I would like to mention the Clemente Estable Institute, whose funding sustained an important part of this research.
I would also like to say thanks to Prof. William H. Sanders and his team from the Center for Reliable and High Performance Computing, Urbana, Illinois for lending us the UltraSAN tool that we used extensively in this work.
All my gratitude to all the people that contributes on the Beowulf discussion list who shares so much knowledge, expertise and know-how. Thank you Raykumar Buyya for the copy of your book “High performance cluster computing”, that was really helpful.
I would also like to thank Sebastián Urrutia and Irene Loiseau who sent us a pre-published copy of your metaheuristic paper [URR1]. You gave us a very good parallel application to work with.
At last, but not at least I would like to thank all the support from my friends and family that sacrificed so many weekends and hours so I could do this. Thank you Olga, Baby and María Eugenia.

1 - Introduction

This chapter will introduce motivations for this research and for parallelism, helping the reader to find out why single processing power might never be enough for all applications. First it will discuss the availability of standard software and hardware, then we will present some commodity-based parallel initiatives and after that we will bound the objectives of this research.

1.1 - Generals

The Supercomputer concept is evolving rapidly. There have been many recent successful experiences of commodity-based supercomputing that have proved –beyond doubt- that Beowulf and Beowulf-like [BEO1] clusters are the way to go when considering price/performance ratio. Free and open operating systems are the generalized choice. Mass-market forces led companies like Intel and AMD to the high performance processor market. Each processor inside a personal computer has capabilities and performance similar to that of a highly expensive scientific workstation. Even though mass market computer industry still runs on 32 bits processors, 64 bit commodity processors are showing up. Free open source 64-bit operating systems are already there and projects like K42 are preparing supercomputing on that platform. The gap between cutting edge supercomputers and commodity ones is getting increasingly smaller.
On the other hand, Microsoft was able to make the phrase “Windows everywhere” real. Microsoft Windows is the *de facto* personal computer operating system. Microsoft operating systems are growing older and a little bit more mature version after version. There are good software development environments and many professionals working for the Windows market. Another key factor that is important is the growing presence of Linux in the operating system market. Even though it is a free operating system it counts with many adepts from the academic side and the companies are starting to use it without shame. In the last years, there has been a significant movement towards the cut of software licensing expenses and Linux plays a key role in this effort. Companies like IBM, Oracle, Computer Associates, etc. are offering services and support for their products also in Linux, and promoting Linux as an “as good as others” platform.
A significant percentage of all the personal computers have x86 compatible processors and use Windows operating system. The server market counts also with a high proportion of x86 processors, but Linux is an alternative to Windows. Linux got an important place within the server market and is very common to find environments with Linux servers feeding Windows clients. The object of this study is to present some models that help evaluating the performance of parallel clusters based both on Windows nodes, Linux ones and heterogeneous environments. Regardless of the fact that this was the configuration in mind and the one used during the tests, nothing prevents the usage of the presented models on other environments.

At the beginning of the nineties, Thomas Sterling and Donald Becker coined the term "Beowulf". According to the accepted definition [BEO2]:

- Beowulf is a kind of high-performance massively parallel computer
- It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks
- built primarily out of commodity hardware components
- running a free-software operating system like Linux or FreeBSD
- interconnected by a private high-speed network[1]

From the very beginning a cluster running other proprietary operating system, by definition, is not a Beowulf, thus it can only inherit a part of all the technology developed under that effort. Nevertheless, it can inherit everything what is related to hardware: processors, network interfaces, memory, etc.; thus, the theoretical performance of any cluster based on the same hardware is potentially the same, if we only consider hardware power.
With this –naive– thoughts on mind, we started working towards the development of a model that can estimate, compare and predict the performance of a cluster based on commodity operating systems.
There are many free available tools used both on commodity and proprietary parallel systems like PVM, MPI, HPF, CORBA, LiPS and RMI. They are both tools for implementing parallel paradigms and they define *de-facto* parallel standards.

1.2 - Commodity-Based Parallel projects

Several initiatives exists that try to harness the power or commodity-based equipment to solve different problems. Some of them try to use idle CPU cycles of computers all over the world, other try to reduce supercomputing costs, etc. In this section we will present a few of this initiatives. The numerical data presented here will always be outdated as this is only a snapshot of one day, so please refer to the original authors of this information.

SETI@home

> *"SETI@home is a scientific experiment that harnesses the power of hundreds of thousands of Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI)"* [SET1]. The way this project works is distributing an application that runs as a screen saver or a background task that periodically gets some work unit, solves it and returns some answer to the server. The task itself consists of searching for particular electromagnetic signals within a certain region of the spectrum called "the water hole", from 1.42 to 1.64 GHz.
> The SETI@home project does one of the most detailed and finely tuned searches ever attempted. Each computer will "listen" to the sky for signals as narrow as 0.07 Hz.
> Data is recorded on high-density tapes at the Arecibo telescope in Puerto Rico, filling

1 High-speed here does not refer to any particular network technology. At the beginning, plain Ethernet networks were considered *high speed* ones. Current *state of the art* defines *high speed* as Gigabit Ethernet or even 10 Gigabit Ethernet. At least three orders of magnitude in network speed separates original beowulf clusters with current ones. As in many other technological situations speed concept is state-of-the-art dependent.

about one 35 Gbyte DLT tape per day. Because Arecibo does not have a high bandwidth Internet connection[2], the data tape must go by snail-mail to Berkeley. The data is then divided into 0.25 Mbyte chunks (which we call "work-units"). These are sent from the SETI@home server over the Internet to people around the world to analyze.
Current statistics are quite astonishing. There have been a total of 1.388.514 users by 11/7/2000 that submitted 149.496.680 results. They performed 298.993.400.000.000.000.000 floating point operations during 330662,52 years of aggregate CPU time, an average of 4,4 Gflops per user or an aggregate of 12,33TFlops worldwide. If we go to the Top500 list [TOP1] and get the system on top, we get the ASCI Red [ASC1], from Sandia National Laboratories. The theoretical peak performance of that system is of about 1,8 Tflops, only 15% of what SETI@home is doing. There are many things to say. ASCI Red is a supercomputer, while the bunch of Internet-interconnected computers that compose the SETI initiative are not. ASCI Red has 9,326 Pentium Pro processors inside each of them capable of performing up to 193 MFlops.

RC5-64

25th September 2002 the distributed.net organization announced that they accomplished RSA Lab's RC5-64 challenge [RSA1] on 12 August 2002. Using the key `0x63DE7DC154F4D039` the text “some things are better left unread” is produced from the encrypted message. The method applied is brut-force. It took 1.757 days and 58.747.959.657 work unit tests so as to find the right one. A grand total of 15.769.938.165.961.326.592 keys were tested, at a mean speed of 103.883.000.840 keys/s by a community of 331.252 participants worldwide.
As it is read from distributed.net's press room announcement: “the RC5-64 project clearly demonstrates the viability of long-term, volunteer-driven, internet-based collaborative efforts.”

Loki

In September 1996, Loki’s architecture and price were presented [WSB1]: $51.379 for a 16 processor system with 2GB of memory and 50 GBytes of disk space. Using that system, between April 25 and May 8, a N-body simulation code using 9,7 million particles ran continuously, with no restarts. The entire simulation of over 1000 timesteps performed a total of 1.2×10^{15} floating point operations, approximately 1.03 GFlops.

Something can be said out of this. The fastest supercomputer in the world in 1999 and third one in 2002 is the ASCI Red, is built with the same family of processors than most of the personal computers of the world (more than 85%). That means that the technology involved is the same. On the other hand, most of them lay on desktops and most of their CPU cycles are

2 At least by year 2000.

idle. It is fairly common to see offices with tens or hundreds of systems, each of them, much more powerful than those used in Loki. There are many differences, but there is a substantial computational power unused there. Is it possible to use it for useful tasks? Can we push the COTS (Commodity of the shelf) concept one step further? Can we adopt not only the hardware, but also the software? Of course, we are not building a Beowulf cluster. Maybe we can make a MSwulf system.
Microsoft is already sponsoring multiple universities to research on parallel scientific clusters over their platform. August 5 2002, the CTC (Cornell Theory Center announced a U$S 60: agreement with Intel, Dell and Microsoft to develop and deliver CTC High Performance Solutions over four years [CTC1]. On the other hand, commodity parallel initiatives keep pushing with Linux. Los Alamos National Laboratory is buying (september 2002) a $6 million, 2,048-processor Linux supercomputer to run its nuclear weapons simulation software.

1.3 - Scope and work environment

As it was presented in the preface, this research is motivated within the initial scope of the ParEnO initiative, a broader initiative that intend to generate a multidisciplinary group of people with several interests and approaches to parallelism. The goal of ParEnO is to generate a task force on the area of parallelism at the service of Uruguayan´s National University (UDELAR), the Engineering Faculty and the industry. Within the scope of that ambitious goal, the objective of this work is to gain knowledge and objective mathematical tools on the cluster construction discipline. We identified the cluster building stage as an important stage that is sometimes not fully considered when building a parallel machine out of commodity components. Generally speed is associated with CPU MHz, but that is not always true. Adding a faster CPU in a network congested scenario does not always prove a wise decision. Experienced system administrators and parallel programmers know all this concerns, but we found no tool adequate for overall parallel system performance analysis. This does not mean that do not exist good system performance evaluation tools and works. There exist excellent papers and works, but they focus on general system performance, comparisons of memory or disk technologies or on algorithms. When we tried to apply this techniques to particular problems and technologies we found that their usage is cumbersome: they generally do not provide an overall system model including algorithms and they mostly emphasize some aspect of the performance. We devised the need of a higher level mathematical tool that is less bound to fine performance details but provides an overall system and algorithmic model. This approach does not replace other models. The other way round. It complements them. Generally other models help taking particular decisions but not overall decisions. The aim of this study is to be able to provide the researcher building a cluster, tools that unambiguously help him to determine which decisions are better for his particular application more than the general benchmarking problem.
At the beginning of current research, several parallel taxonomies were analyzed until we found one that fitted our needs of one that is not specialized only on the hardware or software and that is versatile enough to model properly most parallel scenarios. Chapter 2 presents the results of that research: the taxonomy chosen for this study.

The mathematical model selected for performance modeling are the Stochastic Petri Networks. The success of Petri nets is mainly due to the simplicity of the basic mechanism of the model, which on the other hand present drawbacks on the description of large systems. Several authors extended the basic Petri net models introducing the notion of time. Timed Petri nets can be used for quantitative performance analysis of systems. When random variables are used to specify the time behavior of the model, timed Petri nets are called stochastic Petri nets (SPN). The complexity of the Petri networks we have to work with determined the need for a tool that allows Petri net simulations of several types. Multiple tools (both commercial and academic) exists that represent and solve Petri networks. During 1999 we found 44 different tools for Petri net simulations, 30 of them were free of charge or had some academic discount. We had to discard many of this tools because they did not offer Stochastic modeling or solvers needed. After an extensive research we selected the UltraSAN tool [USAN], which we used extensively all through the research for simulation and as a drawing tool for the diagrams presented in this work. We gratefully recognize the value the tool provided to us.
After finding an adequate taxonomy, a mathematical abstraction that is good for performance modeling and a tool we built performance models for each class of the taxonomy. Chapter 4 presents this generic model templates for each parallel problem of each class, how to build them and how to compute performance estimations that help deciding which is the best option for a particular problem. The model templates presented describes how to build a Petri net that models almost any parallel problem generically, and particularly shows how to determine two performance estimators selected. This performance estimators that can be computed out of the resolution of the network provides objective performance measures that can be used to compare parallel machines.
After describing the theoretical model templates we analyze two real parallel application examples with those templates to show their intended usage. The examples and their application is carried out on chapter 5. The examples are analyzed in depth showing how to benchmark relevant factors, how to apply models and how to predict performance estimators for the execution in certain parallel machines. Afterwards, the estimations are compared to real parallel execution in these machines.

1.4 - Summary

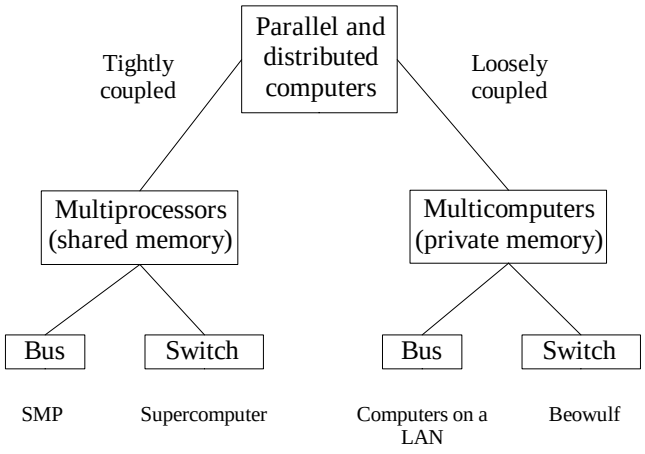
This chapter presented a brief overview of the reasons for using COTS clusters in parallel computing, a few examples of successful parallel applications and problems solved using this approach. In the last years a great deal of effort was invested developing software for harnessing the computational power of all kind of parallel systems. Many advances and knowledge has been gained through the experiences, but there is still a considerable amount of research to perform in the next years to exploit parallelism even more, to gain deeper knowledge of performance improvements and also to develop more automated tools and compilers to abstract the underlying hardware. In the following chapters we will try to contribute with the parallel community proposing a set of model templates that help in the task of system performance evaluation. We will also introduce their usage through instantiation of two templates with real parallel applications.

2 - Taxonomy and Parallelism

The chapter is divided intro three sections and it accurately defines the target and the scope of this work. The first section bounds the set of parallel systems modeled from the point of view of the hardware and the grain of the parallelism studied. The second section introduces the taxonomy for classifying the space of parallel applications and the last section introduces the notation that we used throughout this work for the Petri networks and the modeling and simulation tool that we use, the UltraSAN.

2.1 - Problem determination

When we talk about parallelism, implicitly we refer to a group of CPUs cooperating to a common task in some way. Many different schemes for describing parallel systems have been proposed, but none of them was widely accepted and the concept has evolved through time and technology. One of the most used is the one proposed by Flynn (1972). Flynn selected two characteristics to classify systems: the number of instruction flows and the number of data flows. The characteristics have deep roots in the basics of Von Neumann architecture. The Harvard architecture, implemented in many of current microprocessors, separates the cache memory in two disjoint and independent areas: one for the instructions and one for the data.
Flynn divided both flows in single and multiple, thus, there are four groups of machines: single instruction single data, single instruction multiple data, multiple instruction single data and multiple instruction multiple data[3].
Flynn's approach does not get too deep inside parallel systems: it only identifies them. On [TAN1] one step further is taken and a division of MIMD machines is done. From there, we took the following figure:



3A good description of the categories can be found in [TAN1]. It is important to note that single processor personal computers are still SISD machines even though their microprocessors have parallel execution units: they guarantee the semantics of a standard processor. Processors with MMX extensions also present the semantics of a SIMD on particular operations.

Beowulf systems, the examples presented and the target of this study fall within the loosely coupled branch. SETI@home is a clear example of a extremely loosely coupled system, and the connection speed of systems there is, in most cases, many orders of magnitude slower than the speed of a LAN. Beowulf systems use a wide variety of interconnection devices and technologies, like channel bonding of standard Ethernet NICs, proprietary Myrinet, hyper-cube topologies, etc. to speed up the inter-node communication speed, reducing communication latency.
Not all algorithms have the same characteristics. Their codes have different levels of granularity and parallelism can be exploited at different levels. Starting from the data paths where multiple signals travel in parallel, we get to CPUs where multiple functional units execute in parallel different instructions. Multiple I/O functions that not collide themselves can be issued to different devices like a SCSI bus, DMA operations, etc. Tasks are allocated to CPUs in SMP systems and also over nodes on a cluster.
Levels of parallelism can also be based on pieces of code that can become parallel. The basic idea behind most methods is to avoid idle CPU cycles. Is the basis of multiprogrammed systems: while one task go to wait state (any I/O request), the scheduler allocates the CPU to another task of the ready queue. It is also the idea that leads to the instruction pipeline inside the processors.
On the following table, we present how parallel pieces of code is exploited at different levels:

Grain Size	Code Item	Parallelized by
Very fine	Instruction	Processor
Fine	Loop/Instruction block	Processor/Compiler
Medium	Standard one-page function	Programmer
Large	Program-separate heavyweight process	Programmer

Parallel processing is important at every stage and level, because huge programs have repetitive blocks that are executed many times, and invoke standard one-page functions, which have several loops or instruction blocks. This observation created a tremendous wave that, since the eighties, changed the way computers are conceived. This observation changed the way processors are designed (RISC), and the way the locality is exploited. We will briefly present some of the problems and the way they are addressed.

Fine and very fine grain

It involves the processor, because it is where instructions are executed. The goal is to have the pipeline always out of the stalled state. It is important to avoid hazards, to be able to feed it with data and instructions, to have enough functional units available. Many techniques like prefetching, out of order execution, register renaming, branch prediction, etc. have been used for many processor generations now.

Even though it is not directly related to parallelism, it has a key role in performance: memory bandwidth. If we think about a 32 bits RISC 600MHz processor that can execute –without blocking- 4 instructions per cycle, we have a monster that requires a memory bandwidth of 8.9 GB/s . With a memory bus of 100MHz, the memory system

can deliver only 400-550 MB/s[4][INT1], at least one order of magnitude slower than the core processor speed.

Most of our code has a property called locality: certain portions of the code are executed repeatedly before achieving the result. With a compiler that is smart enough to keep needed results in fast memory (i.e. processor registers), significant speedups are obtained.

This is not possible when large amounts of data and instructions need to be brought to the processor. Now it is important the amount of data that can be accessed by the processor. To increase memory bandwidth, memory hierarchies are introduced and different levels and types of cache memories show up. This helps to speed up problems in which processor registers are not enough, but loops do exist. Depending on which cache has a valid reference to the memory, the number of cycles it takes to retrieve it: from one cycle to as many as 40.

Compilers must be able to help the processor to exploit parallelism at instruction level. Many techniques like register renaming, fetch-ahead, branch prediction only give significant speedups when the compiler takes full advantage of them. The compiler has to make the best out of the code.

Medium grain

There are different programming models and tools that help the programmer deal with parallelism in the code, but it is the compiler the one that deals with it at the block level. Different techniques are applied here to achieve locality and to parallelize the usage of the different functional units. Loops are adapted to increase the accuracy of the branch prediction algorithms, instructions are interleaved so as to avoid pipeline stalls, function calls are replaced in-line to avoid context switches and to maximize locality, etc.

This level can be –and should be- in most cases exploited automatically by processors and compilers alone.

Large grain

Here is where programming tools and models are used, and it is the target of our study. Here the programmer does not deal with instruction in a pipeline, but distributing parts of the problem among different CPUs. The main concern at this level is to find an adequate way to partition the problem, to communicate the parts and to get the result. The programming models help to give foundations partitioning the problem so as to get the result and the tools makes parts of the work easier.

2.2 - Problem Definition and Taxonomy analysis

We will focus our study on large grain parallelism on loosely coupled multicomputers interconnected with high-speed switches. We will build our classification of parallel systems on standard programming paradigms, skeletons and tools. Amongst all possible classifications of

4 Synchronous DRAM (SDRAM)
- •Synchronous with system bus
- •Supports 66 MHz and 100 MHz bus speeds
- •400-550 MB/s bandwidth

the programming paradigms, we considered useful for our analysis to mainly characterize the parallelism by two factors: decomposition and distribution of the parallelism [RAY1]. We will base our classification and taxonomy of parallel programming paradigms according to:

- Task-Farming (or Master/Slave)
- Single Program Multiple Data (SPMD)
- Data Pipelining
- Divide and Conquer
- Speculative Parallelism
- Hybrid models

Task-Farming (or Master/Slave)

In the task-farming paradigm we can identify two entities (group of entities): master and slaves (or for performance reasons, group of masters and groups of slaves). The master is responsible for decomposing the problem into small tasks, distributing them among the slaves, collect results and assemble the problem solution. Slaves perform a simple sequence of steps: get a message with the task, process the task and send the result to the master. In most cases, there is no communication among slaves.
This kind of problems are easily scalable (adding more slave CPUs) and their speedup is quasi-linear. The bottleneck that might arise at the Master is solved making farms with master servers.
The work is statically decomposed and dynamically distributed.

Single Program Multiple Data (SPMD)

It is the most commonly used paradigm. Each process executes basically the same code on a different portion of the data. Due to the division of the problem data among available processors, it is also referred as *geometric parallelism*, *domain decomposition* or *data parallelism*. The decomposition is usually ground on regular geometric structure of underlying physical problems, thus allowing uniform distribution of data among processors. Each processor would need to communicate with its neighbor whenever its calculation needs information held on the neighbor's memory. It might be necessary to provide further synchronization periodically among processors. The communication pattern is usually highly structured and extremely predictable. The data might be self-generated or read from some storage. There is a highly dependence to the processors, and the loss of one of them leads to deadlock states.
The work is decomposed and distributed statically.

Data Pipelining

It is based on a functional decomposition of the parts of the algorithm that are capable of concurrent operation. It is a lower level approach in which different processors execute a small part of the whole algorithm. The communication pattern is simple: data flows in one direction among adjacent processors and it can be completely asynchronous. The efficiency is directly dependent on the ability to balance the load across the stages of the pipeline. The robustness against reconfiguration can be achieved providing multiple independent paths across the stages. It is mostly used in data reduction and image processing.
The work is decomposed and distributed statically.

Divide and Conquer

This approach is widely known in sequential algorithm development: a problem is divided into two or more subproblems, each solved independently and their results are combined to give the final result. In most cases, small problems are just small instances of the original, rising recursive solutions. In parallel divide and conquer, the subproblems can be solved at the same time, given sufficient parallelism. Because the problems are independent, no communication is necessary between processes working on different problems. There are three generic operations: splitting, computing and joining[5].
The work is decomposed and distributed dynamically.

Speculative Parallelism

It is used when it is not possible to obtain parallelism using the previous models. Some problems have complex data dependencies, which reduces the possibility of exploiting parallel execution. In this cases, an appropriate solution is to execute the problem in small parts but use the some speculation or optimistic execution to facilitate the parallelism. Another use of this paradigm is to employ different algorithms for the same problem; the first one to give the final solution is the one that is chosen.

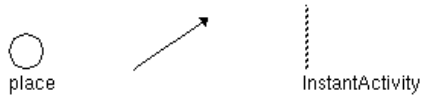
Hybrid models

Real applications not always lie exactly within the definition of the previous groups or, in some cases, it is useful to mix different elements of the different paradigms. They are not generally found on small applications, but in situations where it makes sense to mix them in different parts of the same program.
The distribution of the work and its decomposition is problem dependent.

5Task farming can be seen as a slightly modified, degenerated, one level form of divide and conquer.

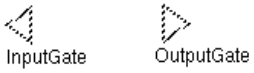
2.3 - Performance modeling abstraction and tool

The mathematical model selected for performance modeling are the Stochastic Petri Networks and the tool used is UltraSAN. Petri net were introduced by C. A. Petri en 1962. The theoretical grounds for Petri nets theory have been deeply investigated and today build a formal structure with a well assessed theory and a broad range of applications. The success of Petri nets is mainly due to the simplicity of the basic mechanism of the model, which on the other hand present drawbacks on the description of large systems. Several authors extended the basic Petri net models introducing the notion of time. Timed Petri nets can be used for quantitative performance analysis of systems. When random variables are used to specify the time behavior of the model, timed Petri nets are called stochastic Petri nets (SPN).
The SPN capability of describing simultaneously both parallelism and synchronization will be exploited all over this work when modeling parallel problems and algorithms. Even though the nature of this work is mainly theoretical, a tool was used not only to draw our networks, but also to test the models with real examples. After researching multiple available tools for working with SPNs we selected the UltraSAN tool, from the Center for Reliable and High Performance Computing, University of Illinois. This tool developed by Prof. William H. Sanders et. al. proved very important in our research. UltraSAN provides a graphical editor for the networks and a set of solvers that allows general problem resolution. UltraSAN also provides some extensions to the standard and stochastic Petri nets which will be used in this work.
The UltraSAN model shares the basic elements with the standard Petri nets: *places*, *arcs* and *transitions* represented with the following graphical elements:



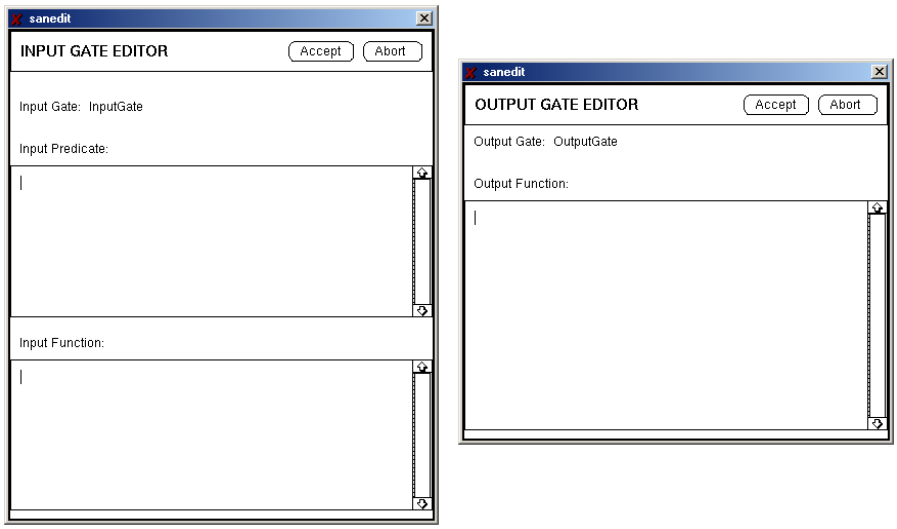
Time modeling is introduced with the *timed transitions*, represented with the following graphical variant of an instant transition:

UltraSAN introduces the *gate* concept. Gates are used to control token movement and logic associated to transitions. There are two gates defined, input and output gates, represented with the symbols:

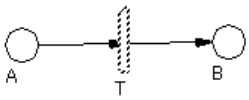


The gates we use in our models are input gates. Input gates need to be connected with arcs to

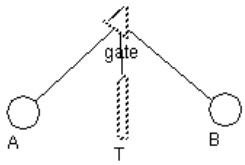
all places considered and also to the transition that is governing. Time parameters are represented with the transition while token movement and conditions that fire the transition are modeled with the input gate. The logic of the gates is expressed with the following dialogs:



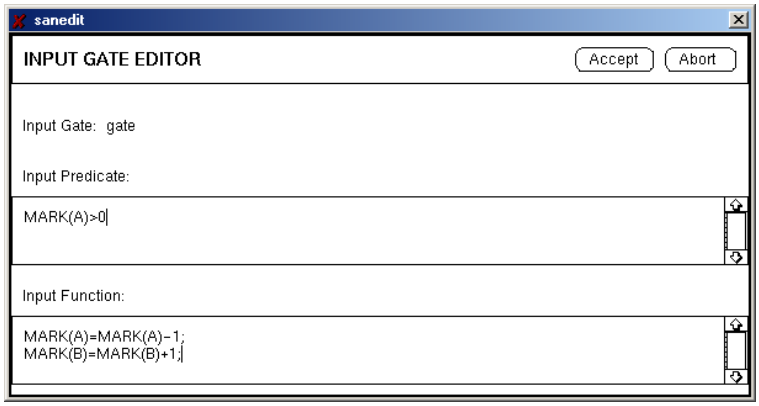
As the tool translates all statements into GNU C, the syntax and possibilities are C ones. The expressions can use specially named macros to refer to defined places, the number of tokens present in them, etc. Detailed information can be found in UltraSAN's documentation.
UltraSAN is an extension of a standard Petri net, thus, all Petri networks are UltraSAN networks. The following simple example helps understanding the input gate extension. We will use the input gate to explicitly code the movement of tokens in a timed transition. Lets call `A` and `B` to a couple of places and `T` to a transition that moves tokens from `A` to `B`. The standard Petri net representation for this scenario follows:



We will place an input gate, named `gate`, replace Petri arcs with connectors to the gate and a connector linking the gate and the transition:



With this representation all token movement depends on the gate coding. Our predicate indicates that we must have at least one token in place `A` so it can be moved. The function indicates that one token is removed from `A` place and one token is inserted into `B` place.



The macro `MARK(<place>)` returns in runtime the marking of the place `<place>`. The value can not only be inspected but modified. We will exploit several of this capabilities in the rest of the work. An issue that has proven important is the ability of drastically reduce the space state of a problem using gates. In situations where *n* tokens have to be moved from one place to another, the use of a single gate that moves all tokens at once turns a state space of *n+1* states into a 2 states one.

2.4 - Summary

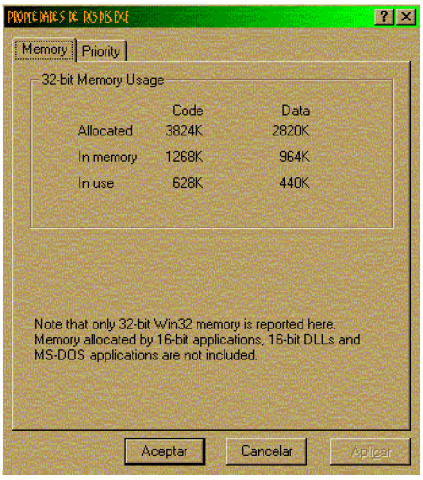
This chapter has defined the scope of this study within the field of parallel systems and applications both from the point of view of the systems involved and from the point of view of the application or algorithm used for solving the problem. We will focus our analysis on the exploitation of large grain parallelism using loosely coupled parallel systems. This chapter also presented a taxonomy for the classification of parallel algorithms that considers the distribution and decomposition of the work statically or dynamically.
We also presented here the formal structure (SPNs) used for modeling parallel machines and algorithms that will be used in this work. Also UltraSAN tool and the basis of its notation were introduced in this chapter.

3 - Modeling Distributed.net's RC5

We will present in this chapter many concepts used in the rest of the work, using the distributed.net (http://www.distributed.net) RC5 project as a case-study. We will try to model the problem ranging through different scales, discussing the trade-of of detail vs. abstraction at every stage introducing Petri networks that models each. The reader that is already familiar with performance indexes, parallelism, modeling with Petri Networks and execution simulation can skip this chapter and continue reading from the following chapter on. Reading of this chapter is also useful for understanding the kind of decisions, trade-offs and considerations used not only in the modeling but in the practical examples analyzed in the following chapters.

3.1 - Introduction

When the RC5-DES 64 bits encryption scheme was launched, a challenge was presented by the RSA: a message encrypted with such technology was posted and a prize of U$S 10.000 was offered for the group that decodes the message[RSA1]. Many groups started working on this challenge. One of them, known as distributed.net addressed the challenge using idle CPU cycles from internet connected systems. They offer an executable that can be easily installed on computers ran by people willing to share their CPUs with distributed.net people. They will get 20% of the prize if their CPU is the one that finds the key to the message.
The whole Distributed.net's RC5 project is based on the idea of parallelizing the task of testing the key-space. Current electronic technology and processing speeds does not allow a single CPU computer to solve it within a lifetime. The method chosen for solving the RC5 - 64 bits challenge is the brute force, that is, to try each and every key in an orderly fashion. A server distributes “packets” or collections of work units to machines that request them and wait for the answers. The job is done when a client finds the key.



The whole space of solutions is divided into 68.719.476.736 (2^{36}) work units with 268.435.456 (2^{28}) keys each. A computer based on a Intel Celeron Mendocino processor (CPU family 6, model 6, stepping 5) of 466 MHz and 128 KB of L2 Cache can try 1,2 millions of keys every second while editing text. Such a machine can exhaust all the keys within the next 490 centuries without overclocking.
The distributed.net's RC5 client software (dnetc or rc5des.exe) runs on multiple systems. On Windows systems it has a really low impact on the user perceived performance of the system. The task is highly CPU-bound and there is really very little I/O associated with it: retrieving work units and sending results over the internet and storing the work done locally. The whole application fits in RAM and allocates quite little memory, thus, adding very little overhead on the memory and IO subsystem. It runs with the lowest priority available on the system. The combination of

this factors, and the lowest priority available on the OS, makes this application have no perceivable impact on an interactive system, thus, is designed to run all day long as a background process. On a system running this application, there are almost no idle CPU cycles.
The problem itself is rather peculiar because all the experiments are independent and the results are only dependent on themselves. No communication needs to take place between clients, only between the client and the server. Further more, no previous information needs to be considered for any calculation. The only two states that have to be considered are if the solution has been found or not.

We are going to use a simplified RC5 system as a model to study and discuss the impact of different parameters on the problem.
We will use the same pseudo-code as a simplified algorithm that describes the RC5 system:

Master

```
while not solution-found
       wait for connection;
       send message;
       receive results and store;
       send blocks to test;
       close connection;
```

Slave

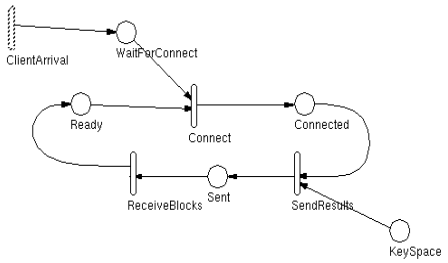
```
While not solution-found
       connect to key server;
       receive message of the day;
       send results;
       receive blocks;
       solve blocks;
```

We will assume that the server is absolutely devoted to its task and the systems that run the RC5 client software are used for interactive (user) applications during the day. We are going to use Petri Nets to model the problem.

We need to find an adequate model not only for the client and the server but for the integrated system. We need to find an adequate unit to represent both the time and the work done by the system. We should not choose it too small, because the number of states in our system would burst; we should not choose it too coarse, otherwise we would lose too much detail in our model. We will try to determine the most appropriate unit for this problem.

The server model

We will start presenting a simple net that models the behavior of the server:



The tokens that will represent the “state” of the server can be in three places: `Ready`, `Connected` and `Sent`. They will be “waiting” in `Ready` place until a client makes a request and

after that, it will receive (when available) the portion of the problem solved by the client and will send a new part of the problem for solving. The name of these places are meaningful from the client's point of view.
The problem of concurrent access of many clients can be modeled using multiple tokens on the ready place. The number of tokens on the ready place represent the maximum number of concurrent connections that the server accepts. The time spent on `SendResults` and `ReceiveBlocks` transitions should depend on the number of tokens currently present on the system.
The place `KeySpace` contains a token for each block to be solved, a total of 2^{36} tokens, that will be adapted to the client work units on the client's subnet. In this way we represent the evolution of the solution: as tokens are consumed from the `KeySpace` place, the key-space is being exhausted. The client arrival will be modeled on the following paragraphs.

The client model

The work-unit given by the CPU instruction cycle

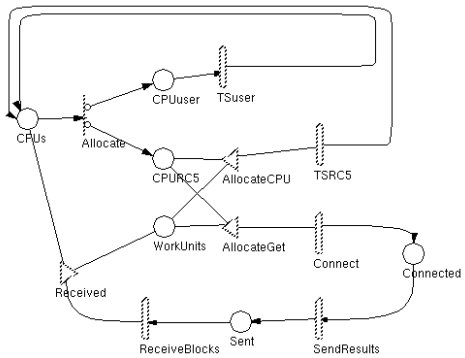
The smallest unit that we can choose is the instruction of the processor where the client software runs. Even if we assume that the instructions require a fixed amount of time (the x86 family remains CISC), this is a far too small unit and leads to a great deal of problems even in the case of our simplified model like idle CPU cycles, pipeline stalls due to cache misses, page faults due to swap-outs and context switches, etc. We prefer to have a coarser measure of the performance of each individual system like the MFLOPS or MIPS, which tries to give a figure that summarizes these and other aspects like memory bandwidth, cache size, etc. Even if we would like to consider this unit we would find some problems with all the CPU-cycles that have to be considered. If we assume a 500 MHz processor, we have approximately $5x10^8$ cycles every second, $3x10^{10}$ cycles every minute, $1{,}8x10^{12}$ cycles every hour, $4{,}3x10^{13}$ cycles every day and $1{,}8x10^{16}$ cycles per year that each system is running. If we assume approximately 50.000 systems working daily on this problem[6] we should consider $9x10^{20}$ CPU-cycles for every year of work. Far too much.
We can see this problem if we compare the order of magnitude of our single task unit and the time it would take a single machine to linearly solve the problem: $2{,}15x10^{-9}$s and $1{,}5x10^{13}$s respectively. The difference is of 22 orders of magnitude.

The work-unit given by the OS timeslice

A logical aggregation of CPU-cycles is given itself by the operating system as timeslices, that is, the maximum amount of time that the CPU is exclusively allocated to a particular task by

6 The number of systems that helped with the challenge each day can be found in distributed.net's web site.

the operating system. In a system that is running mainly user jobs like a word processor, there are very few processor interruptions other than those generated by the system timer. A fast operator can generate bursts of 280 keyboard interruptions every minute that go directly to the keyboard processor queue and then can be retrieved by the processor. If all the keystrokes would generate an interruption in the processor, there would be several millions of CPU cycles to handle the character, do the context-switch and allocate the CPU back to the RC5 code. The system timer interrupts the processor many times every second to run the system scheduler. The *quantum* of time that the processor is assigned to a thread depends on the flavor of Windows that the system is running [RUS1], but it varies from approximately 7 to 15 ms. Assuming that nearly all the interrupts come from the system timer we can think of a model in which we allocate the processor fixed amount of times to different tasks.
Based on empirical measures, we found that, for office environment desktop computers, the CPU is most of the time allocated to the idle task: 90% of the time or even more. On a system running the rc5des.exe program, the idle CPU time is given to the rc5des.exe program. We represent that giving 90% probability of allocating the CPU to the `CPURC5` place and 10% to the `CPUuser`.
We should estimate which portion of the block is solved in a timeslice so as to "consume" tokens from the place `WorkUnits` and represent completion of the block. Doing that, we can divide the solution of the problem into a certain number of allocations of processors to the RC5 task. The key server would distribute parts of the problem and each CPU on the network will consume them.[7]
With this approach, we have skipped many details of the system, and we can get a higher level of abstraction. Within our work-unit, we collapsed many factors of the system like the instruction-set of the processor[8], memory bandwidth, size of caches, etc. and we get an overall indicator of the low-level system performance that describes which portion of the problem can be solved during that period of time. For a theoretical analysis we need to use measures like MIPS, MFLOPS or other indicator of expected system performance where our algorithm would run. This is a good model of a single system, but in the real case there are many different systems with different performance indexes. We must group them into classes of equivalence according to the speed they solve RC5 blocks, study a representative and model the interaction of the classes, biased with the cardinal of each class of equivalence.
The following figure presents a model for the client:



7 This would be true having a network with homogeneous machines. In the case of Internet and the real RC5 project, there is a great variety of systems. The analysis remains valid making different classes of systems, each of them, with equivalent processors.
8 e.g. Availability of MMX extensions

We are using UltraSAN [USAN] as a drawing, modeling and simulating tool for Petri Nets thus, we adopted the input and output gates as a tool for controlling arbitrary changes in the marking of places. The use of gates replaces inhibiting and multiple arcs, replacing a graphical notation with expressions and formulae.
There are two input gates, `AllocateCPU` and `AllocateGet` that are used within the RC5 algorithm to decide if we need to fetch more work units[9] or we still need to compute more. The activation predicate for `AllocateCPU` is:

```
(MARK(CPURC5)>0) & (MARK(WorkUnits)>0)
```

which means that we allocate the CPU to a `WorkUnit`. The function of the gate is to decrement the marking of both places in one. In this way, consuming tokens from `WorkUnit`, we represent that we have solved another part of our system.
The activation predicate for `AllocateGet` is:

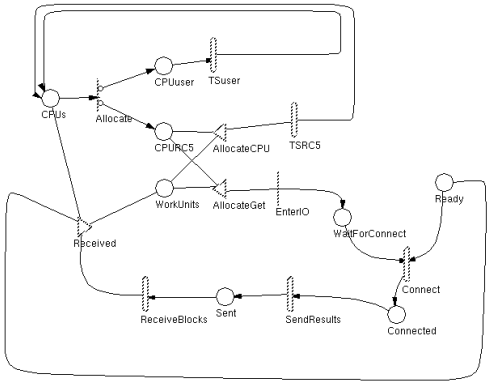
```
(MARK(CPURC5)>0) & (MARK(WorkUnits)==0)
```

which mean that we cannot allocate the CPU to a `WorkUnit` because we need to fetch more. The function of the gate is to decrement the marking of CPURC5 in one. In the real case, the CPU would issue the “connect” primitive, return to the scheduler, switch to “waiting for I/O place” and would be re-allocated to another ready task. We are not trying to represent the whole allocation algorithm, but the allocation of the CPU to the parallel application. That is why our simplified model of the OS remains valid.
There is only one output gate in our system: Received. It receives a token from the server, returns the CPU to the scheduler and puts the retrieved work units in the “to do” queue.
The definition of the gate is as follows:

```
MARK(CPUs)=1;
MARK(WorkUnits)=MULT;
```

The factor MULT depends on the processor speed, the timeslice of the OS and has to be either empirically or theoretically estimated. We define MULT, for the general case, as the number of timeslices needed to solve an individual work unit.
The other factor that has to be determined is the probability that has the CPU to be allocated to a user task.
In the following figure, we introduce a Petri Net that shows the interaction of the server and one client:

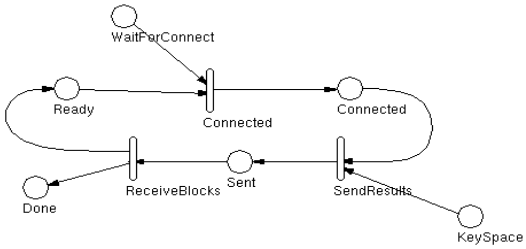
9 In this case our work units are different from the blocks distributed by the RC5 server. Our work units represent the average number of timeslices the CPU is assigned to a block so as to solve it.



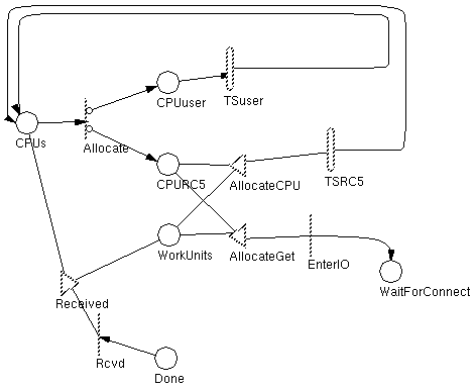
Since the token leaves the place, wait for connecting until new blocks reach `WorkUnits` place, the client is mostly blocked for I/O and the dominant factor in the analysis is given by the token evolution within the server model.

So as to take advantage of UltraSAN's composed model feature, we need to introduce an auxiliary place: `Done`. The purpose of such state is to have two states in common for the client and the server, so as to detach them, but also, to be able to combine them with the modeling tool.

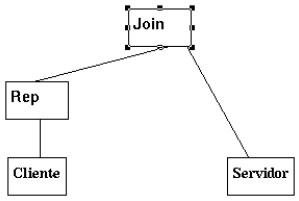
The following figure shows the final model for the server:



Every time that a token leaves the Sent place, a token is moved to `Ready` and another to `Done` places. The following figure shows the resulting model for the client:



The following figure represents the composed model:



These models remain valid considering that all the computers have the same numerical power. A better model can be represented modeling the space of computers with different classes of equivalence. The members of those classes provide the same MIPS to the RC5 challenge. With the UltraSAN this can only be modelled defining different Client models like Client01, Client02, etc. and joining them.
Even though we have developed a full model for the application, the state-space of this problem is once again far too big: even if it is theoretically possible to model a system this way, it is not numerically adequate for calculation purposes. As mentioned before, a computer based on an Intel Celeron Mendocino processor (CPU family 6, model 6, stepping 5) of 466 MHz and 128 KB of L2 Cache can try 1,2 millions keys every second while editing text. On average, it gets 110 to 120 timeslices every second, thus, it can try approximately 10 thousand keys every timeslice. As we have noted before, a block consists of 2^{28} keys to be tested or, what is the same, 26.850 CPU timeslices (around 4 minutes of CPU time). As the key-space was divided into 2^{36} blocks, the model should consider at least $1{,}8x10^{15}$ states representing the evolution of the problem. The space-state is too big to be analyzed, and a new aggregation needs to be used.
Once again, if we compare the order of magnitude of our single task unit (in this case, a timeslice) and the time it would take a single machine to linearly solve the problem: $8{,}33x10^{-3}$s and $1{,}5x10^{13}$s respectively. The difference is of 16 orders of magnitude.

A few kilobytes are transmitted each time communication needs to take place and the whole communication process takes no more than a few seconds even over a slow line. A practical example is taken as a reference. The transmission required to obtain a group of 10 RC5 packets took only 17 seconds using a 33.600 bps modem. 2.775 bytes were transmitted during the session (1.306 bps), including the DNS query. That very small amount of information was enough to communicate the key server the results of the work done during the last period of time and to retrieve work for the next four hours. The relation between processing time and communication time is very high: around 800 times, nearly 3 orders of magnitude.
At this point we can depict some performance limits on the RC5 architecture. RC5 relies on TCP/IP as a transport/network protocol suite and connect to the not so well-known-port 2.064 on the key server. There is a limit of 65.536 concurrent connections to one port due to the 16 bit socket identifier (handle). This limits how many machines can simultaneously fetch keys because standard Berkeley sockets cannot deal with more than 2^{16} clients talking to a server on one port. Handling 65.536 clients is a very important load on any system. The memory overhead on the operating system would be significant and all the operations related to the network would experience important delays.
Another notorious problem comes with the need of bandwidth: 1.306 bps are necessary for each concurrent client retrieving keys to test. If the server is using a T1 line, only about 1.150 clients can retrieve keys simultaneously, or what is the same, $1{,}65x10^{7}$ machines can fetch keys

once every four hours. This is a limit to how much we can paralellize this problem: adding more machines would only lead to waiting without processing until a connection can be made to the key server. Anyway, if we are able to put $1{,}65x10^7$ Celeron 466 MHz machines to work together, we would be able to solve the RC5 challenge in 10 days, 18 hours and 48 minutes. This "virtual machine" would achieve a crunching speed of 19.8 Tkeys/s, each key involving tens of integer instructions.

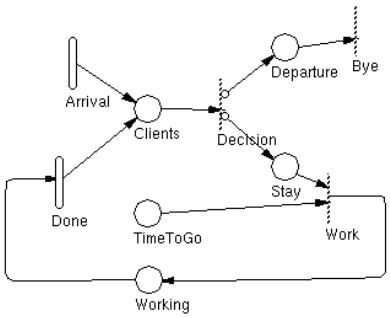
If Moore´s Law stays the same for the next twenty five years, the whole RC5-64 bits challenge could be solved with a M^2COTS cluster with 32 Intel Pentium XV processors of 40,32 Tkeys/s each. It would take no more than four hours to exhaust the key-space and could be a nice examination for a student to solve a crypted message given by the lecturer.

The work-unit given by the solution of a RC5 packet

Another meaningful aggregation is given by the way in which the problem is divided and distributed to the clients: the time for solving RC5 packets. Each packet consists of a group of one or more work units, generally eight. A typical connection to the key server retrieves 60 work units in 8 or 9 blocks to be solved, that is about $1{,}61x10^{10}$ keys to be tested. At a rate of 1,2 million of keys per second, that gives us about 4 hours of Celeron-crunching between transmissions. The RC5 client can be configured in many ways, according to the Internet connection available. We will model hosts with permanent Internet connections.
With this new quantum of problem-solving, we are taking a coarser approach: we are representing a time evolution that is $1.7x10^6$ times bigger[10] than the previous one or, what is the same, we are representing the solution of a set of blocks as a whole, instead of keys. We are not concerned with OS details or what is the user doing, but with the average time the CPU was assigned to the RC5 client, and thus, the time spent solving a block. As in the previous case, the clients will fetch blocks from the key server, solve them and return the results but we will not model details within the client. We will consider only a couple of different CPU usage profiles on the client.
The first usage profile considered is the *idle system*, that means, a machine already booted, either with a user logged in or not that is not doing any batch task like disk optimization or virus scanning. As its name says, this profile represents a system that is doing nothing but running rc5des.exe. The only interference to a 100% rc5 dedicated system is caused by the scheduler overhead.
The second usage profile considered is the *interactive system*: a system whose primary task is to run user processes, generally, with higher priorities than rc5's priority. The consequence is that the CPU is given to the rc5des.exe process only after all the user's CPU needs are fulfilled: no other task with higher priority can be in the ready queue if the rc5 is to be scheduled.
If we plan not to break the solution of blocks into smaller work units, we need to represent the different profiles in a way that is independent of task execution interleave, level of user activity or virus detection. We must collapse all these factors into a simple and handy unit of work evolution.
Lets choose any processor as a reference processor and use it to turn solution-space remaining

10 This factor is calculated with reference to the 466 MHz Celeron processor, but will vary on different systems: the OS's timeslice is independent of the numerical power of the processor.

into time-to-exhaust-solution-space using a theoretical dedicated system based on the reference processor. It is clear that both representations model the evolution towards the solution of the problem. Using the time evolution approach it is simpler to represent different CPU contributions: different CPUs with different loads can consume tokens faster or slower than our reference processor according to their speed and load.
As we saw on the previous section, the time spent on communication for retrieval of new work units is almost 3 orders of magnitude smaller than the time needed for solving the whole packet, and can be omitted.
The following figure represents a model based on systems delivering approximately the same CPU time to the RC5-64 problem.



The system has three main places: `Clients`, `TimeToGo` and `Working`. Like in the previous models we have a place whose tokens represent the evolution of the keyspace to be checked, in this case, `TimeToGo`. The place `Clients` collects the idle clients ready to work.
There is a timed activity, `Arrival` that models the arrival of new CPUs wishing to cooperate and depends on publicity and other social issues. The clients face regularly the decision of continuing working or leaving the project. That is the meaning of the instant activity `Decision`. The reason why a client leaves the challenge could be many, like the frustration of so much CPU hours and so little reward or simply forgetting about the challenge one year later when the hard drive had to be formatted. The `Decision` could lead to `Departure` or `Stay` places. As soon as a token gets to `Departure` it is removed from the system: `Bye` instant activity.
For clients that still wish to solve the challenge, the `Work` instant activity removes a token from `TimeToGo`, allocates it to the client and puts it in the `Working` place. The tokens are removed from the `Working` place as soon as they spend there a certain amount of time, equivalent to the one needed for our reference processor to solve our work unit.
This is valid for homogeneous processors with similar system load, but it is not clear how to handle the variety of processors available with diverse load.
We will make classes of equivalence within the Internet hosts space and give different number of representatives to each class. Based on a reasonable inverse linear behavior of the key-space solution speed with respect to the CPU load on equivalent systems, we can say that a system with a CPU that is equivalent in performance to our reference processor will spend double the time with a system load of 50% of the CPU time assigned to other tasks than the rc5 software. A system with double CPU power will need half of the time if it gives its 100% of CPU to the rc5 task, and so on[11].

11 This is not totally true due to the impact of context switches in the overall system performance caused by missing locality within the processor's complex cache systems and other performance factors.

With this perspective of CPU time assigned to the rc5 problem, we can classify systems running the rc5 client according to the time they can give to the problem resolution depending on load and CPU power. Each of these classes would have different number of members.
We noticed that this last aggregation is not suitable again for performance prediction using Petri nets for the same reason: excessive complexity of the numerical solution. With a level of granularity of approximately four hours it is not possible to model OS details and small factors of interactive tasks but it is also too small with respect to the whole solution time. Once again the order of magnitude of our single task unit compared to the order of magnitude the time needed to solve the whole keyspace is quite too big: 1^{12} and $6{,}87x10^{10}$. The difference is 10 orders of magnitude.
Even though it could be possible to try further aggregations of the problem, we believe that they are not meaningful and they cannot give richer information about the evolution and predict the performance of the system than the arithmetical calculation depicted recently.

Processing speed determination method.

We have so far tried to model the complete resolution process and we were able to produce accurate models that are numerically far too complex. In the previous section we mentioned the possibility of some kind of analytical resolution based on the concept of “resolution speed”. We will use this concept for the problem resolution.
This approach differs from the previous one, as it does not try to model the complete problem resolution, but to determine which portion of the whole problem can be solved within a certain amount of time. Then it is straightforward to determine the time it will take to our system to fully solve the problem, provided that the estimated “speed” remains constant for the whole process.
As we mentioned before, we will partition the space of CPUs according to their approximate contribution to the problem resolution and estimate the number of CPUs in each class. Knowing this two figures in all cases we can estimate the contribution of each class to the problem resolution and consequently. The construction of a Petri network for this purpose is straightforward as is very similar to the last one built, but it is not interesting in the current context. Lets assume that we are modeling a RC5 work unit with our token, then, the token consumption speed for each equivalence class is a direct function of the number of elements in the class and the estimated speed of each element. Now we have the token consumption speed of each class, then the overall token consumption speed is the sum of individual class speed. This single figure is our token consumption speed and should be an upper bound for the real system speed, as we are not modeling any kind of delays blocking, etc. that would lead into unused clock cycles, and thus, slower overall resolution speed.
In this case the analysis of the Petri net that we would have constructed suggested a particular analytical way to estimate a particular measure, the system's processing speed. Using this measure is possible to determine the time it would take to the system to exhaust the tokens that models problem's complexity. In this way we have simplified the system, as we do not have to model the whole resolution, but we can estimate it out of an intermediate estimation.

12 We have chosen our unit as the time needed for our reference processor to solve a work unit.

3.2 - Conclusions of RC5 modeling

During the last paragraphs we proposed several Petri Net models of the RC5 system that generates models with different level of accuracy of the system but that they all impose practical problems to the resolution: in all cases the number of states in our Petri Net grows beyond what we can handle or would like to use to predict the system performance. It makes no sense to use a system for prediction that is more complex and more inaccurate than the real system. We were not able to break the problem into pieces that are small enough to keep rich details about the problem and also that are big enough so as not to make the number of states in the Petri Net be reasonably bounded.
As we tried through the different levels of aggregation we were able to identify key interactions of the client and the server that helped us understand why this problem can be solved using a highly de-coupled set of computers like the Internet: absence of interaction among clients and the huge difference between the time spent processing and the time spent sending results and retrieving more work.
We were also able to settle, at least, some theoretical and practical limits imposed by the operating system, Internet and communication state-of-the-art.
We finally presented an arithmetical way of predicting the performance of the system that distributed.net is using to address the RC5 challenge. With simple arithmetic it is possible to calculate numerical throughput and time needed to exhaust the solution space. Anyway, in such an uncontrolled system like the Internet, a key issue to answer the time needed to solve the problem is a social issue: how many people would like to lend their CPUs to distributed.net.
Another problem that this particular simulation faces is the technological evolution. It took almost 5 years to solve the challenge so, using Moore's law the industry doubled the performance 3 times, so state of the art the systems that helped in the last period of time were eight times faster than the ones that started the challenge. This kind of long term simulation faces evolution problems of the hardware, specially in highly heterogeneous and uncontrolled system like Internet. It is far beyond the scope of this work to model this kind of long term system evolution. The models that we will present in the following chapters assume constant the system performance during the problem resolution.

4 - Model templates for general parallel applications

In the second chapter we presented the fundamental reasons why we divided the set of parallel problems into five groups: ***task farming***, ***single program multiple data***, ***data pipelining***, ***divide & conquer*** and ***speculative parallelism***. During the following sections we will discuss the main issues that have to be considered for each group and we will depict the way in which a model can be built so as to represent a specific instance of a program of a parallel group running on a particular system.
It is a well known fact that the performance is not exclusively system dependent. The software that runs on a particular system plays a key role on the performance evaluation of the system. Efforts like SPEC rely on statistical analysis of which different kinds of codes run on the average system. The people that compare their system with SPEC agree with those generalizations of the average software execution profiles. A higher SPEC index does not guarantee that a particular software will be faster on the new system. That is why we do not concentrate only on the topology and performance of the hardware but also on the software structure, inter process communication, etc.
We have done an exhaustive analysis of a master-slave process when we analyzed the RC5 model in the previous chapter. We will generalize what was done and introduce parameters that were omitted due to peculiarities of the RC5 problem scale.
When representing a system, we must choose the level of detail to be included in our model. Too little detail means that there will be important characteristics of the real system that will not be included in our model. Too much detail means that we will need to measure and calibrate many parameters of the model; also, the numerical methods available for computing interesting performance measures from the model may be too slow or too imprecise to be useful. Unless we consider particular cases in which the parallelism is highly exploitable, a lot of effort has to be placed when building the model in order to ascertain its validity.
One useful technique is to introduce small variations in the data to test stability of the model and to learn about the tolerance of the model to changes.
Finally, if the results of the simulation are not conclusive or the complexity of the model precludes its numerical solution, it might be useful to make a prototype of the system and run it. This prototype usually would include more detail than a numerical model, in order to provide better information about the real system.

4.1 - Performance Indexes

Another important point that must be decided before building models for each of the classes of parallel problems is to choose a set of performance parameters or indexes to be evaluated. The need for these indexes is to have concrete meaningful, problem independent indicators of the system performance, that have deep roots in both the software that will run on the system and the system itself.

This has important effects on the models we will build, as a model appropriate for evaluating a steady state parameter of the system may not be useful to compute transient performance measures.

We have chosen to study the following parameters:

a) The Total Execution Time (TET): is the execution time of the problem, from the initialization phase until the process reaches its ending. This is a transient measure, which is particularly important in real-time or quasi real-time parallel systems, but it is generally important on every system that we code: we would like to know how long it will take to find a solution to our problem.

b) The Mean Execution Speed (MES): if we can measure the problem size in some work unit (say for instance number of floating point instructions, blocks to solve, etc.), we can think of an important parameter which is the speed at which our system (measured in work units per time units) solves the problem. This allows us to estimate the system's processing capacity. We define MES as the size of the problem (represented in work units) divided by the TET .

We will discuss later foundations for these definitions, specifically for the second one. We will see that under specific situations one measure can be easier to determine than the other. Furthermore we will see that in many cases, if we can estimate properly the MES and we determine the size of the problem, then we can estimate the TET directly from the MES definition and viceversa.

According to the MES definition, we can represent it using the following equation:

$$MES = \frac{WorkUnits}{TET}$$

We will be interested on systems in which the TET can be divided in the next sequence of stages:

a) initialization,
b) regimen,
c) ending;

and their associated times: T_i, T_r and T_e. When we introduce the concept of a “regimen stage” we are facing the idea of a stationary phase, which would lead us to the problem of defining properly what is a stationary phase in our particular networks. This is difficult to determine in the general case of our problem, specially due to the fact that we move tokens from a initial place to a final place. If we think of infinite times and we follow a scheme similar to that one presented when studying the RC5 problem, we find that the stationary phase would consist of the state in which all tokens are moved into the final absorbent place. Our concept of “regimen stage” differs from the stationary phase mentioned before and refers to the constant problem solving phase found between the initialization and the ending. We will not try to formalize more this concept for the moment.
Assuming the previous considerations, we can express the Total Execution Time as the sum of the Initialization, Regimen and the Ending Times like:

$$TET = T_i + T_r + T_e$$

Replacing this equation on the MES definition we get:

$$MES = \frac{WorkUnits}{T_i + T_r + T_e}$$

Our study will focus on situations where $T_r >> T_i$ and $T_r >> T_e$, thus, it is valid that $TET \simeq T_r$. We can conclude that:

$$MES = \frac{WorkUnits}{TET} = \frac{WorkUnits}{T_i + T_r + T_e} \simeq \frac{WorkUnits}{T_r}$$

Using any of these measures, we may study if it is possible to obtain any speedup with a parallel execution scheme replacing a single processor one, to obtain expected order of execution time and the numerical speed of our system solving the particular problem. We will use the terms system, cluster and group of CPUs basically interchangeably because we are not tying ourselves to a particular hardware configuration, even though the base of this study is commodity components. The analysis can be applicable to a set of interconnected uniprocessor systems, interconnected multiprocessor systems, NUMA machines, etc.

We will present a way of modeling each problem within the five groups discussed and how to obtain the general performance indexes that describe the numerical performance of a system conformed by certain hardware solving a particular problem. We will depict which parameters have to be estimated so as to model the system, but it is impossible to generalize how to estimate each parameter for each parallel group. Each time that a particular system is modeled, it has to be decided how to estimate and calibrate the constants needed for the model to predict properly the behavior of the system.

4.2 - Task-Farming (or Master/Slave)

In the task-farming paradigm we can identify two entities or groups of entities: *masters* and *slaves*. There may be only one master or a group of them; there may also be different groups of slaves, but essentially, we are facing the same problem that might have been partitioned due to performance reasons. Either way we will model a virtual uniprocessor master serving a farm or group of slave systems.
The *master* is responsible for decomposing the problem into smaller tasks, distributing them among the slaves, collecting results and assembling the problem solution. It is important to note that all this means overhead. Decomposing a problem, even picking up ranges without performing any processing to the data set results in extra administrative work. Keeping track of which parts were allocated to which slaves and, depending on the situation, deciding that

there was a problem with that slave (maybe the answer did not arrive within a certain acceptable amount of time) and allocating the same part of the problem to a different slave means extra CPU operations that have to be considered as overhead.
Slaves perform a simple sequence of steps: get a part of the problem, process it until a solution is found and send the result to the master. In most cases, there is no communication among slaves.
This kind of problems are easily scalable (by means of adding more slave CPUs) and their speedup is quasi-linear as there is little or no interaction among slaves. The bottleneck that might arise at the *master* is solved (if possible) making farms with master servers. If we apply a state-of-the-art layer 7 traffic redirector, the logic and intelligence that the slaves need to choose the appropriate master can be collapsed within a hardware device that can become a commodity component soon. State of the art network manufacturers are including redirection and load-balancing facilities into their boxes. Layer 7 traffic redirecting devices have a monetary cost similar to multiple slave nodes together and the problem can be solved with more intelligence on the software at the master nodes.
We will represent both processes with a high level pseudo-code that makes clear the interaction between both the master and the slave, leaving problem complexity bound to `process`, `partition` and `assembleSolution` functions. The pseudo-code follows:

```
master
    receive P  (size n=|P|)
    partition P into P1, P2, .... , Pn
    k=1
    repeat asynchronously
           {send Pk; k+=1}
           {receive Rj; process(Rj)}
    until #{Rj}=n
    assembleSolution
    fin
```

```
slave
    repeat
           get Pi
           Ri = process(Pi)
           send Ri
           until  !∃ Pi
    fin
```

The partitioning of the problem is a major issue, not only for its inherent difficulty but for the consequences on the solution process. It is important to note the difference between the process partitioning and the modeling work-unit election: the first one is a design decision that deals with the problem resolution while the work-unit election summarizes what we consider relevant and what we can abstract so as to model the system.
The asynchronism of the repeat loop in the master represents the fact that there is no particular sequence of `send` and `receive` in the general case. We expect that the master process delivers *n* pieces to the slaves and collects *n* solutions from them. Maybe the master creates a thread for each P_j allocated to a slave, maybe sends all $P_j s$ and then waits for the results, etc.
The collection of results could include some fault tolerance on the clients. The master can set timers when delivering problems to clients, so if a time-out condition arises, then the problem is allocated to another client willing to solve it. There is no problem if multiple solutions are returned for a given problem, only one is stored.

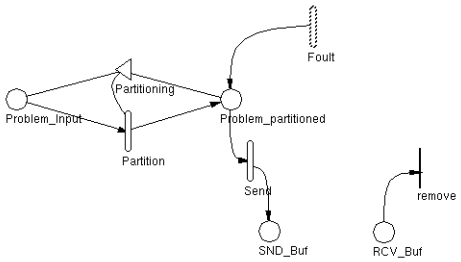
System modeling

We base our Petri net model on the previous pseudo-code. We are representing a set of CPUs

that will act as slaves that will loop until there is no more work to be done and a master that dices the problem into adequate pieces and delivers them to slaves.

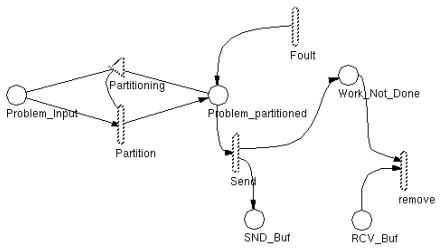
Masters model

The following net represents the master process:



The initial configuration of the network is with all places empty but the `Problem_input` one. A token is placed there representing the initial problem P that is going to be solved. There is a timed activity, `Partition`, that represents the main initialization task: partition the problem into the tasks that will be solved individually by each slave. We use the input gate defined by the UltraSAN package to manage the extension in the functionality of a regular timed transition. Instead of removing one token form the `Problem_input` place and placing it into `Problem_partitioned` place, we place *factor* tokens into `Problem_partitioned` place. Factor is a global variable which we use to set different initial conditions in our simulations. Varying this factor, we can modify the size of the problem to be solved by each slave. It generally has a correlation with the time in which an individual task is completed, which is represented in the slave model.
The `Problem_partitioned` place holds all the tokens that represent the problem partitioned and the `Send` timed activity delivers pieces to slaves ready to work. The distribution function for the `Send` activity represents the time involved in the process of accepting a connection and sending the necessary information for the slave to start working. Generally this function summarizes operating system, network availability and communication issues. It is possible to place an input gate that pauses the `Send` activity if more than a certain number of tokens are already placed in the `SND_Buf` place. This is useful to achieve a closer representation to reality and to bound the number of different states on the Petri net, which is useful for the numerical resolution of the problem.
The timed activity `Fault` represents the lack of completion of slave tasks due to hardware problems, blackouts, system crashes, etc. As there is a high independence among slave tasks, the missing one is placed back in the `Problem_partitioned` place. All that has to be done afterwards is `Send` it and allocate it to a slave. Generally, the rate for this activity is extremely small, which represents the rare event of a problem in a slave. Even though the stability of commodity systems today is very high (comparable to workstations a decade ago), when considering a cluster with an important number of slaves or mistakes caused by improper user handling of their systems, the joint probability of a single (any) system crash grows to a level that might be considered. The existence of the `Fault` activity has an important impact on the numerical solution of the theoretical model: even though the rate for the distribution function is very small, there is a non zero probability that an arbitrary big number of failures occur within

a bounded interval of time. The consequence of this fact is that the number of states of the network is unbounded, thus, many of the solving algorithms can not be applied with this activity. If the resolution algorithms need to calculate the state-space prior to the numerical resolution of the networks, then they can not be applied. In this kind of networks simulated results prove adequate.
The results coming from the slaves are placed on the `RCV_Buf` place. We are not modeling post-client processing, thus we only consume tokens without representing any particular activity[13]. There are some cases that can be mentioned here. For example, it is possible that the master needs to communicate something to the running slaves related to the received result: this is the case for example in a branch and bound algorithm when a new partial solution was found. There also exists the possibility that the answer from the slave results in new tokens added to the `Problem_partitioned` place. Either way, the final post-processing changes the completion time with a bound and well known factor. In case that this factor needs to be included in the network, a timed activity `assembleSolution` can be introduced that removes all nodes from *`RCV_Buf`* place when processing is done.
One aspect that has to be considered when building the network is to keep it as simple as possible, while modeling all the relevant information. If the model gets too complex, it will be extremely time-consuming to solve it.
At this point we can note that there is loss of information from the *master*'s point of view: as soon as all nodes have been removed from the `SND_Buf`, it is not possible to determine if the processing is done or some slave is still processing. We can always inspect the slaves to check if any one is still processing, but it might not be practical in all situations.
When asyntotic behavior is analyzed, current network is adequate. When terminating simulations are needed, when a *master-slave* network needs to be integrated into another network or for synchronization means, we need to introduce a new place in the network that lets us keep track of all the pieces of work delivered to the slaves. The following net represents the master process with the new place introduced:

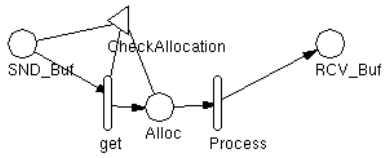


As pieces of work are sent to the `SND_Buf`, they are also copied to the place `Work_Not_Done`. As soon as processing is done, tokens are removed both from the `RCV_Buf` and from the place `Work_Not_Done`. With this new network it is easy to determine if all the slaves finished with their pieces of work. If no tokens remain in `Problem_partitioned` and `Work_Not_Done` places, then all processing is done.

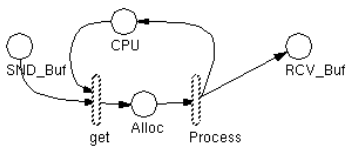
13 We use a instant activity for representing it. Due to constraints in the tool we are using for modeling, we can not have a model running with a instant activity after a place used for joining two networks. When doing real simulations, tokens can be consumed on the slave or a timed activity can be used instead of the instant one.

Slave models

The *slave* model is a little bit simpler, because the *slave*'s structure is also simpler as the logic of the problem resolution resides on the *master*. The main complexity (if any) that exists at the slave is the procedure to solve piece of problem that is allocated to it. The slaves only retrieve pieces of work to be done, solve them and send the results back to the master. The following net represents each slave process:



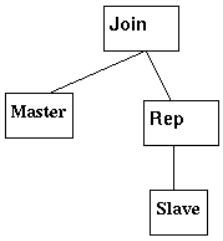
The name of `SND_Buf` and `RCV_Buf` are named from the master's point of view, not from the client's. It is needed due to UltraSAN constraints for joining nets: Places should be named the same on joined networks, and we selected the master's perspective.
The rest of the processing on the client is straightforward: a token is removed from the buffer and allocated. Then the client spends some time processing and then the token is returned to the master. The input gate `CheckAllocation` is responsible for not allowing more than one token inserted on the `Alloc` place because we model the allocation and solving of one piece of problem at a time. It is also possible to model multiprocessor systems or groups of systems with equal performance to be modelled as tokens allowed by the `CheckAllocation` input gate.
Even this network is adequate for our purposes, the design is somehow influenced by the tool we are using: the input gates are an addition to classical Petri networks done by the UltraSAN tool that allows very powerful and expressive operations over a network. What we mean at this point is to move only one token at a time, because our nodes will consume only one work unit at a time.
We can model the same behavior replacing the input gate with a node called `CPU`, initialized with one token. Arcs should connect this node with activity `Get` and the activity `Process` with node `CPU`. With this network we represent the allocation of pieces of work to CPUs. When both a CPU and a work item are available together, they are both consumed. When processing is done, the CPU is released (token returns to `CPU` place and is again allocable) and a token is placed at `RCV_Buf`, indicating that some result was obtained. The following figure represents the network:



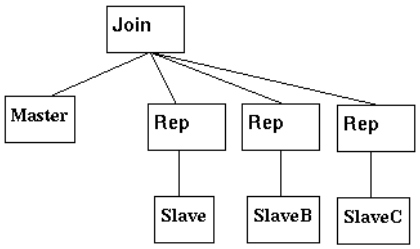
Both networks can be used interchangeably.
It is good to mention now that in the case of multiprocessor systems there are two alternatives:

either initialize as many tokens in the CPU place as processors the system has and add only one slave network to the systems network or to add as many slave networks as CPUs to the network that models the whole system.
The estimation of the processing time is a important factor for the predicting capability of the model. Particular information of the specific system being modeled should be represented at this stage.

The complete system is modeled joining a set of slaves with a master. Unless we face particular sick configurations and problem scales, bandwidth and processing capacity of the master should not be a bottleneck in our systems: processing and communications capabilities should be considered enough, at least, at the beginning. In general, Task-farming problems are more CPU-bound than I/O-bound, thus, CPU is much more a bottleneck than the network speed. Another factor that has to be considered is that raw communication speed is not a problem currently as 10GigabitEthernet is already available. The problem is that the processing overhead to process 10Gigabit (up to 20 million Ethernet packets per second) is too much for a single CPU. Master's burden must be split not because of pipes width, but because of CPU might not be enough if processing at the master is considerable at high network speeds.
We use the following composed model to represent the conjunction of a master and multiple slaves in a single network:



The `Rep` box represents the replication of the `Slave` network. We use the `Join` box to combine a set of *slaves* with the *master*.
This system represents a set of systems that deliver approximately the same CPU power to the resolution of the problem. Heterogeneous systems could be represented either calculation complex distribution functions for the `Process` activity or using multiple groups of slaves, each of them with equivalent CPU power available for the problem resolution. We represent such heterogeneous systems with the following composed model.



Each *SlaveX* network is exactly the same as the others, but differs on the distribution function for the `Process` timed activity. On the replication box, we use different numbers to represent

the different number of instances of the different processing speed.

Performance parameters

We have already presented a method for modeling *master-slave* systems. We still need to provide means that help deciding how to architect the system and how to develop the software. The final goal of the modeling is to determine as early as possible the best way to engineer a certain parallel system, to reduce the complex set of different ways in which the system can be designed, or answer if current technology can address certain particular problem within particular time restrictions..
We will present now how to estimate the MES (Mean Execution Speed) and TET (Total Execution Time) based on the model described before.

Total Execution Time

As we have previously defined, the total execution time of the problem is the time from the initialization phase until the process reaches its end. Based on our system, we can calculate this measure as the time it takes to the system to move all tokens out of the system. This implies that we have to simulate the whole resolution of the system, that means, we have to model the whole system described before, place the tokens in the initial place and run a simulation until all tokens are removed away from the system.
Not all simulation tools allow this kind of estimation. If the tool that is being used allows to calculate network's steady state, we can introduce a variation in the network's layout. We can make a cycle from the final state, the absorbent configuration, to the initial one, so after the processing is done, the network is restored to the initial configuration. The process of restoring the initial configuration must have an associated timed transition with a known amount of time. We can simulate this new network in the steady state and measure the fraction of time that the network spends restoring the initial configuration. From that value we can then estimate the counterpart, that is the TET.
It is not always possible to ascertain this measure basing our forecast on simulation due to the complexity of the numerical solution. It can be the case that it might take too long to calculate the TET out of a complete execution simulation. If it was possible to compute the MES for that system, then it is possible to estimate the TET as the complexity of the problem divided by the MES, that is, how long it will take our system to consume all tokens at the processing average speed provided that most of the execution time is spent on the regimen phase. On average, this estimation is adequate, but it does not consider the behavior before and after the regimen phase. A source of error to this estimation is due to the time the system runs out of the steady state. If the regimen state takes most of the execution time, then the estimation is adequate, otherwise, it has to be specifically considered.

Mean Execution Speed

If we are calculating the MES after the calculation of the TET, then with only an additional

arithmetic operation it is possible to compute MES according to the definition. On the other hand, if it is not feasible to directly compute the TET, then it is possible to try estimating the MES before and afterwards, based on that value, to determine the TET.
As we discussed before, we need to estimate MES. We will estimate the regimen problem solving speed of our system (measured in work units per time units), what we call that estimation MES. This allows us to estimate the processing capacity of the system and also to estimate the time in which the stationary phase of the problem can be attained.
The time in which the regimen phase is attained has to be calculated in some way that is problem dependent. After the system is in its regimen phase, one of several standard techniques (regenerative simulation, batch means, etc.) can be applied to determine the MES. Changes in the network can be done to simplify the determination of the steady state like adding infinite initial tokens or making cycles so as to keep the overall number of tokens constant according to the time evolution. The last alternative is generally preferable for the sake of numerical simplicity.
If the system was modeled according to the previous recommendations, there has to be a couple of states, called `SND_Buf` and `RCV_Buf` that permit the uncoupling of the master and the slave. The evolution on the part of the network that models the master process does not present a steady state behavior: tokens will be consumed from the `Problem_input` place until there is no token remaining.
We can concentrate on the set of slaves that consume tokens from the `SND_Buf` and their induced Petri net, a subset of the original net. We will estimate how fast the slave nodes consume tokens. Even though we can theoretically analyze the processing speed based on an infinite set of tokens in the `SND_Buf` place, it is numerically simpler in our Petri Net to model a finite number of tokens being reinserted after they are processed in the initial place so as to keep constant the number of tokens on the network and also to keep bounded the total number of states of the system. The average number of tokens that cycle the network in a certain period of time should be called MES. It can be determined as the difference between the total number of tokens in the network and the average number of tokens in the `SND_Buf` place. This is equivalent because all the tokens that are not in the `SND_Buf` place are cycling the network accordingly to the definition. The advantage of this alternative is that is simpler to compute the average number of tokens in one place than the number of tokens cycling.
As a rule of thumb, it is important that on the regimen state there is always more than one token on the `SND_Buf` place. If all tokens are consumed, it is possible that a slave is willing to process, but there is nothing to process. If there are always tokens on the `SND_Buf` place, it means that there is always more work to be done than slaves to accomplish it, and thus, there is no idle slave. In that situation, we are solving the problem as fast as we can. Generally it is wise to have more tokens tham CPUs willing to solve pieces of work.
It is a general fact in this kind of systems that the regimen state is reached quite soon. As there is little or no interdependence between slaves, there are generally no constraints that prevents slaves from getting their work. In that scenario, the most likely event that would stop a slave from getting more work to be accomplished is a bottleneck on the server side. It could be of many different types like CPU when it is preprocessing the job, maybe splitting it into smaller tasks; it could be a network bottleneck due to the fact that a high number of clients are all at once eager to get their jobs, and as they were spawned together, they collide trying to access the server, etc. After all the clients get their pieces of work they are all working on they will keep their pace, only interfered by eventual bottlenecks on the server side.
This final estimation of the MES will be greater than the one calculated from the TET as it does not consider the initialization and post-processing times. In a way, it gives the fastest

processing speed that the system can achieve from the slaves point of view.

4.3 - Single Program Multiple Data (SPMD)

It is the most commonly used paradigm. In most cases, the problem suggests how to distribute the problem to each CPU. Each process executes basically the same code on a different portion of the data. Generally the differences are due to boundaries of the space being modeled like the walls of a nuclear reactor, height of a layer of air in the atmosphere in a shallow-water model or the limit of a geographic region considered for the dispersion of pollutants. In all these cases, *something special* has to be done so as to preserve physical constants/values of the system like entropy, energy, mass, etc. For example, when we consider a fine-grain atmospherical model of the winds over a city, Corioli's force is applied to all points of the grid, but the system modelled is not a closed one: the winds entering and leaving the region (differences in atmospheric pressure) have to be modelled in the boundary with functions which evolve in time.
Due to the division of the problem data among available processors, it is also referred as *geometric parallelism*, *domain decomposition* or *data parallelism*. The decomposition is usually ground on regular geometric structure of underlying physical problems, thus allowing uniform distribution of data among processors.
Each processor would need to communicate with its neighbor whenever its calculation needs information held on the neighbor's memory. In many cases, with the model we represent a piece of the universe in a given time t_0 and we use the models to predict how our universe will be at time t_f. To speed-up the calculation of the evolution of our modeled universe as time evolves, we partition the initial state within a set of processors and parallelize the time evolution from t_0 to t_1. After the processing, each processor has computed his associated part of the universe in the time t_1.
Nothing can travel faster than light and all the forces in nature have different strengths according to what is considered and distance[14]. That is why there is a cone of influence implicitly associated to every point of the universe and to what is going on there. Lets think about two points, X and Z that are separated more than $c(t_1-t_0)$, where c is the constant representing the speed of light. Nothing has to be considered in Z from X in the instant t_0 to calculate the state t_1 and viceversa. If two points X and Y are separated less than $c.(t_1-t_0)$ it might be necessary to exchange information between both of them so as to calculate the next state. What defines the interaction within the model is the model itself, what is being modeled, what is considered relevant and what can be obviated. There is no general rule that can be usually applied to determine a fixed set of neighbors.
It might be necessary to provide further synchronization (barriers or other methods) periodically among processors. As the processing is relatively similar between all processors the synchronization is not a waste of time and it can be used for checkpointing, very useful in cases of crash-recovery. On the other hand, it leads to problems mixing CPUs of different power or time-shared systems with different loads because in most cases it will lead to systems with the performance of the slowest CPUs.
The communication pattern is usually highly structured and extremely predictable. According to the problem itself, the data might be self-generated or read from some storage. There is a

14 At least in the four-dimensioned universe that A. Einstein helped us understanding.

high dependence to the processors, and the loss of one of them leads to deadlock states. There has been an enormous amount of work improving reliability on clusters, facilitating process migration between CPUs, fault tolerance, etc., but until now, the best price/performance ratio is obtained on non-redundant systems, fault intolerant.
The modeling of these systems is not as straightforward as the previous case. The communication plays a more important role on these problems and the way the communication pattern takes place within this models determines the way the net's graph lies.
Before the process begins, there is a stage of division of the initial condition of the problem between the processors. The interrelations of the processes are obtained from the model and the communication pattern is known. We can determine the neighbors of each processor, understanding neighbors as two processes that share memory.
There is an initial stage in which the original problem is divided in small parts and the interaction between processes/processors is defined.
We represent that stage with the following piece of code:

```
Initialization
      receive Problem Pt0
      divide Pt0 into {P1,t0 , P2,t0 , P3,t0 , ..... , Pn-1,t0 , Pn,t0}
        ∀ Pj ,1 ≤ j ≤ n , V(Pj) = {Pk1 , Pk2 , .... Pkm}
```

The function V returns the set of neighbors of a given subproblem. We assume the general case in which the neighborliness is reciprocal, and thus $P_i \in V(P_j)$ means $P_j \in V(P_i)$ [15]. Each process itself executes basically a simple sequence of stages, represented by the next piece of pseudo-code:

```
process
      receive subproblem Pi,t0
      set k=1
      repeat
            Ri,t = process(Pi,t)
            for Pj in V(Pi)
                  async send(Pj,Ri,t)
            for (Pj in P) / Pi ∈ V(Pj)
                  Rj = receive(Pj,Rj,t)
            Pi,t+1 = Ri,t ∪ Rj1,t ∪ Rj2,t ∪ .....
            k++
      until k=max (or other suitable condition)
      end
```

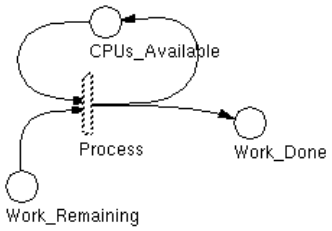
The processes are mainly loops that run until a certain condition is verified. Checkpointing was skipped for the sake of simplicity, but on long runs it is a must. Checkpointing would consist of storing state information in a (maybe safer) permanent storage that can be used for resuming in the event of a system failure.
The process receives the initial condition of the problem to be solved and runs the appropriate algorithm on it, producing a certain result $R_{i,t}$. The result is communicated to the neighbors and neighbors results are received. When we use async send we do not constrain ourselves to a particular routine, but with the general concept that the produced result is sent and local computation is not suspended until the neighbor receives it. The result can wait on the neighbors protocol stack, could be written to disk or can be held until the neighbor polls for it.

15 This has a solid physical ground on the way natural forces operate. If a processor holds information that has to be taken into account when determining the next time-step of a neighbor, it will need the information from his neighbor to compute his own next time-step.

The specific implementation is not a main issue here.
The process of getting neighbors results is blocked because we can not continue computing until we have all the information needed to compute the following step. After we have all the information needed, that means, our result plus our neighbors results, we assemble the data set that is going to be used to compute the next time-step, and so on.
Not all the algorithm implementations of SPMD problems are exactly written this way, and a wide variety of particular solutions exist. The process does not have to have all the information needed to compute the next time step when it starts, it can poll for it whenever it is needed. This *lazy* approach is highly algorithm dependent, because neighbors memory could be required any time. Another problem with not so structured approaches is the problem of deadlock states, which are avoided in a structured design.
What our general pseudo-code represents is that we need information from our neighbors to compute each time-step, and also that we need to share with them some of our own results and so on. The data exchange here plays a key role. Neither our neighbors nor us can compute results without the other, and the data exchange process slows down overall calculation power of the system. It could be the case of a problem in which too much time is spent on communication and a single processor system could perform better due to the lack of communication overhead.
We will represent the system and the data exchange with Petri nets.

System modeling

On first high-level thoughts, we can make an abstraction of this system as a set of CPUs and a set of work pieces. Each CPU gets a piece of work processes it and gets ready for the next piece of work. The network that represents this would consist of one place where all pieces of work are represented with tokens, another place where available CPUs are also modeled with tokens, a timed activity that models the allocation of a piece of work to a CPU and its resolution and finally a place that receives all solved pieces of work. The following network represents this:



Studying this network layout, we can see that we have completely lost all inter-process communication modeling. The `Process` activity simply removes CPUs, but there is no modeling of the fact of available CPUs that can not get a piece of work allocated because information from the neighbor is not available yet. When we want to represent this system on a Petri Net, we find that it is not possible to sketch a single network layout in which we have a pool of CPUs and a set of tasks to be accomplished because we lose the interaction and interleaving of processing and sharing information. In these systems, it is important not only

that processing and communication takes place, but in which order and how much time is spent waiting for synchronization. We concluded that we can not depict a general network layout, as it was done with the *task-farming* class of parallel problems. We present here a procedure for constructing the Petri Net associated to a given SPMD algorithm, the kind we described before. We will have to construct networks for each particular problem.

Let us first introduce the definition of the places and transitions that will conform our net. Each process P_j will basically be processing or waiting for others results. Lets call `init_Pj` to the initial place where the token representing the state of the process P_j is and `proc_Pj` to the timed transition that represents the processing at process P_j.

Each time that processing is done, a token is removed from `work_Pj`. The tokens in the place `work_Pj` represent the remaining work of the current run. After the processing is done, the token goes to the place `wait_Pj`. The token is also "copied" to fictitious places that represent the asynchronous communication between processes. The instant transition `sync_Pj` removes the token from the place `wait_Pj` and puts it again in place `init_Pj` where cycle continues.

We introduced the fictitious places to represent the asynchronous interchange of information. We call `snd_PjPk` to the place used to represent that information sent from P_j to P_k is queuing, waiting to be retrieved by P_k. An instant transition[16], `sync_Pj` is used to continue processing only after the processing of P_j is done and also that P_j's neighbors have sent their information.

The following procedure is used to partially define the Petri Net associated to a given problem:

Let the processes be $\hat{P}=\{P_1,P_2,....P_n\}$ and lets define the function $V::\hat{P}\rightarrow\widehat{P^n}$ as $V(P_j)=\{P_{k_1},P_{k_2},....P_{k_m}\}\forall P_j,1\leqslant j\leqslant n$

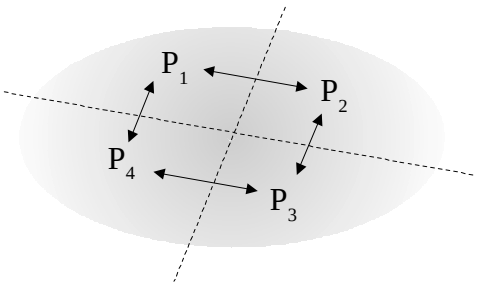
For each process P_j in $\hat{P}$
- add a place labeled *work_Pj*.
- add a place labeled *init_Pj*.
- add a timed transition labeled *proc_Pj*
- add an arc from *init_Pj* to *proc_Pj*.
- add an arc from *work_Pj* to *proc_Pj*.
- add a place labeled *wait_Pj*.
- add an arc from *proc_Pj* to *wait_Pj*.
- add an instant transition labeled *sync_Pj*
- add an arc from *wait_Pj* to *sync_Pj*.
- add an arc from *sync_Pj* to *init_Pj*.
- for each process $P_k/P_k\in V(P_j)$
  - add a place labeled *snd_PjPk*
  - add an arc from *proc_Pj* to *snd_PjPk*.
  - add an arc from *snd_PjPk* to *sync_Pk*.

With the previous procedure we defined the layout of the network, the places, activities and transitions. Some parameters of the network still need to be defined. To have a fully defined network, we still need to determine the distribution functions for the timed transitions and the number of tokens. Before following with the definition of the network, we need to state something about its complexity. The complexity of the network can grow considerably. For each process three places, two transitions and five arcs are added without considering neighborliness, that might easily add four more places and eight arcs for each process. The resolution of the resulting network can consume some CPU power and could take significant effort. The process of constructing such network on a tool proves also difficult. For complex systems it is a good thing to have some kind of automated interface (not only the graphical

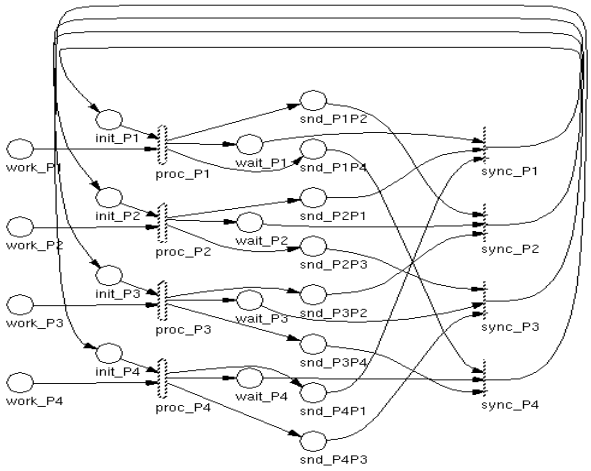
16 It can be argued that synchronization is or is not something instantaneous, as it requires interprocess communication of some kind. In our model, we are not placing the cost of synchronization in this activity. We are only modeling the blocking.

one) that can be programmed for constructing these networks.

Lets apply the procedure to a simple configuration. Our example configuration consists of four nodes corresponding to a domain distribution that divides a space in four areas. The communication pattern is a square. The following figure represents the division in four regions and the arcs between nodes represent the data interchange.



Applying the procedure we get the following network:



The complexity of the network can grow significantly as the number of nodes grow. Current Beowulf projects face hundreds or even thousands of nodes that can lead to extremely large nets.

To continue with the definition of the network, we will determine the number of tokens for each place. Places $\texttt{proc\_P}_j$, $\texttt{wait\_P}_j$, $\texttt{sync\_P}_j$ and all $\texttt{snd\_P}_j\texttt{P}_k$ start with zero tokens. All $\texttt{init\_P}_j$ places start with one token, representing the processor ready to be allocated. Even though multiprocessor systems can be used, multiple tokens can not be placed on the $\texttt{init\_P}_j$ places, as the lack of synchronization problem would arise. We represent each processor on its own, even if it shares resources with others on a SMP. The work to be done itself is represented by the $\texttt{work\_P}_j$ place. We must place there as many tokens as necessary to represent the solution of the problem. The key issue at this stage is to determine a meaningful work unit for the problem.

The rule-of-thumb for SPMD problems is to represent each time-step with a token[17], thus, if we want to calculate the final state of our study universe after 10.000 time-steps, 10.000 tokens should be placed on each $work\_P_j$ place if we want to model the execution of the system until time 10.000. The execution stops when all $init\_P_j$ tokens return to place $init\_P_j$ and $work\_P_j$ places are empty. The execution halts because not all predecessors of $proc\_P_j$ transition are fulfilled.
With the previous kind of simulation, we simulate the whole execution of the system. If we want to calculate the asymptotic state of the system, we would delete the $work\_P_j$ place, and on the resulting network leave the simulation running until it stabilizes so performance indexes can be retrieved.

The remaining aspect of the network is the definition of the timed transitions. At this point something has to be known about the execution times of each time-step. Based on the complexity of the problem, the estimated number of operations and performance indexes of processors, it is possible to estimate the distribution function of each time step for each processor. Lest we have some practical/empirical information about the execution times, normal distribution can be used for modeling. If the prediction should be accurate, a small prototype of a real execution of one time-step might be coded and measured.
If we take an analytical approach to the performance prediction, we estimate the complexity of the problem and use benchmark figures to predict execution times, we will get one figure: average/expected execution time for each loop/work-unit. From strictly theoretical analysis we will not get variances or other indicators. After that we can discuss if our system will run on a dedicated set of machines or on interactive systems. If we run on dedicated machines, that do not execute regular intensive administrative tasks, then the only interference comes from the operating system, which can be considered constant for work-units comprising more than a few seconds. On those cases deterministic execution times can be chosen for modeling. This would lead to simple systems and the prediction can be considered optimistic. On the other hand, if the system provides different amounts of CPU times to our process due to any reason, we have to estimate the execution time based only on one figure: the expected execution time. As we only have the expected “mean” execution time, but no variance or other value, the exponential distributions appears both as simple and pessimistic, due to its inherent variance. On most cases we can consider that the times predicted using the theoretically estimated times with exponential random variables is a worst-case bound for the real system execution times. The expected execution time should lie between both estimations. Better accuracy can be achieved prototyping.

It is important to note that there is a high interdependence among processes and processors. Let's suppose that two adjacent processors[18], A and B, have different performance (maybe they belong to different processor generations, have different clock speeds or belong to different manufacturers with different design technologies) and they need to exchange information before computing the next time-step due to boundary calculations. Lets suppose that A works twice as fast as B. This means that process A will finish its calculation, send its results to B and block itself waiting for its neighbors, particularly B, results before continuing calculating. As B is about half of the execution time, A will spend about the other half of B's execution time

17 If the number of time-steps is too big, maybe each token represents multiple time-steps. If we take this approach, much care has to be taken because we miss individual blocking/interleaving of processes.
18 Not physically adjacent, but respecting to blocking. We consider two processors adjacent if they share information for their results.

waiting for B to complete computing. During that interval of time A's processing power is either allocated to other tasks or is wasted idle looping or twiddling its silicon thumbs. It will not be possible to take any profit of A's speed for out problem in this situation.
Furthermore, if we extend this reasoning to all processors in a run, we see that if one processor is faster, there will be no benefit, as it will wait for its neighbors. Even worse is to consider the effect of having n-1 fast processors and 1 slow processor: after some time-steps, all the processors will be waiting for the slow one, and will have no effect on the overall performance. The speed of the system will be bounded by the slowest processor, the weakest link.
It is possible to think about distributing the size of the data set of the problem assigned to each processor, but it is not easy to manage heterogeneous processors. Even if we can divide the regions allocated to each processor according to its computing power, the impact on the complexity of the communication pattern and the coding is generally not worth.

Performance results

We have already presented a method for modeling SPMD systems. We still need to provide means that help deciding whether it is convenient the parallel execution vs. the single processor one. The final goal of modeling is to determine as early as possible the best way to engineer a certain parallel system, to reduce the complex set of different ways in which the system can be designed, or answer if current technology can address certain particular problem.
We will present now how to estimate the MES (Mean Execution Speed) and TET (Total Execution Time) based on the model described before.

Total Execution Time

Based on our system, we can calculate this measure as the time it takes to the system to move all tokens out of the system. This implies that we have to simulate the whole resolution of the system.
If the system was modeled accordingly to our recommendation, then there exists a set of states named $work\_P_j$ where tokens representing the amount of work to be addressed by processor P_j are placed when the simulation begins. Let's call T_j to the number of tokens corresponding to the partition of the whole problem that is allocated to the process j. The system has to be simulated until the places work_Pj run out of tokens, which means, that all the work allocated (T_j tokens) to them is exhausted. Successive terminating simulations could be run to determine the approximate elapsed time until the execution ending (batch means) if the simulation tool does not determine how long it takes to reach the absorbent state.
Not all simulation tools allow this kind of estimation. If the tool that is being used allows us to calculate network's steady state, we can introduce a variation in the network's layout. We can make a cycle from the final state, the absorbent configuration, to the initial one, so after the processing is done, the network is restored to the initial configuration. Simply adding a timed activity that monitors all $work\_P_j$ places and when they all get empty simply places all initial tokens back, we get a network that does not fall into an absorbent configuration. The process of restoring the initial configuration must have an associated timed transition with a known amount of time. We can simulate this new network in the steady state and measure the fraction

of time that the network spends restoring the initial configuration. From that value we can then estimate the counterpart, that is the TET.
There is a theoretical lower bound for the TET that could be computed from the resulting network after removing all the $snd\_P_iP_j$ places and their associated arcs. The resulting network is the junction of *n* models of different uniprocessor systems without connection running independent processes. In this particular case, we have each processor consuming tokens at the speed given by their processing capability, represented by the timed transition labeled $proc\_P_j$. For each processor, and according to the distribution function associated to proc_Pj activity we can compute the average processing time A_j . This calculation is problem dependent and there is no general rule. If the distribution functions were calculated already, then the only thing to do is to apply the appropriate formula. Then, for each processor *j*, the average execution time would be estimated multiplying the amount of work times the average time for accomplishing it: A_j x T_j
The total execution time for this system would be:

$$max\left\{ A_j \times T_j \right\}$$

We will now explain why this is a lower bound for the original system. When we removed all $snd\_P_iP_j$ places and their associated arcs, we removed all the interrelation among processors. Specifically, that means that we stopped modeling all the time intervals in which every processor is idle, but it can not continue computing because they have to wait for adjacent processors to share their information. The original system models this information also, thus, it can never be faster. In the particular case that it is never necessary to wait for a neighbor, the TET of both systems would be the same.

It is not always possible to ascertain this measure due to complexity of the numerical solution. It can be the case that it might take too long to calculate the TET out of a complete execution simulation. If the modeled system presents a stationary behavior and it was possible to compute the MES for that system, it is possible to estimate the TET as the complexity of the problem divided by the MES, that is, how long it will take our system to consume all tokens at the processing average speed.
On average, this estimation is adequate, but it does not consider the behavior before and after the stationary phase. A source of error to this estimation is due to the time the system runs out of the steady state. If the stationary state takes most of the execution time, then the estimation is adequate, otherwise, it has to be specifically considered.

Mean Execution Speed

If TET calculation was possible, then the MES calculation can be done just by applying the definition. If this was not the case, it is possible to estimate a value for the MES in this kind of networks. In the following section we will describe how this is done.
If the system was modeled according to the previous recommendations, for each processor P_i there has to be a state called $init\_P_i$. The $work\_P_i$ place holds the tokens that model the problem space. While evaluating this measure, we are not interested in the whole problem itself. We can obviate this places, and thus, the whole problem evolution. Generally this is the case as we are not calculating the MES after the TET. We will study the sub-network obtained from the removal of $work\_P_i$ places. In the resulting network, tokens only cycle as fast as the interlocking permits. Each cycle of a token represents the completion of a work-unit, that

means, if we are able to count the number of cycles that all tokens perform within a certain period of time, then we know the number of work units that can be solved on that period of time. We can count the number of cycles associating signals to a specific place (i.e `init_Pi`). Afterwards, using the batch means method we can estimate the MES.
It is a general fact for this kind of systems, that before reaching the regimen state the system has to cycle many times, generally more cycles than states. Depending on the level of interdependence among slaves, there are generally no rules that describe how to propagate the delays among slaves. This is the main reason while it is not possible to mathematically formulate the accumulation of delays interleaved with the processing. In that scenario, the most common reason for idle CPU time is the need for neighbor data. If we analyze this recursively, one process could be waiting for data from a neighbor who is also waiting for data from another neighbor, who is also waiting for data from another neighbor, and so on. This could be as deep as the whole number of processors. In the worst case, it is possible that *n-1* processors are waiting for *1* processor. In a controlled situation this situation is very rare or even less improbable, but if the algorithms applied by each CPU has a high variance, it is possible to have many idle CPUs per time interval.
The accumulation of this effect is the reason why the estimation of the TET described before is a lower bound for the real TET. All the combined effect of this makes the MES calculation difficult.

4.4 - Data pipelining

The data pipelining paradigm is based on a functional decomposition of the problem, in which different tasks of the algorithm are identified which are capable of concurrent operation. Each processor executes a small part of the total algorithm. Each process corresponds to a stage of the pipeline and is responsible for a particular task. The communication pattern can be very simple, since the data flows between the adjacent stages of the pipeline mostly in only one way, thus, this paradigm is sometimes referred as data flow parallelism. If we face a *pipelinable* problem, that means, separable in sequential stages, each with a relatively high computation-to-data ratio, it is possible to build a pipeline with different stages on different machines.

As in all pipelines, the efficiency is directly dependent on the ability to balance the load across the stages as the performance is bounded by the slowest stage. If there exists a stage that is considerably slower than the rest, and there is no dependency between consecutive tasks in the pipeline, we can allocate multiple processors (as much as necessary) to that intermediate task, that will work in parallel, so as to obtain similar execution times on all stages. Another workaround to this problem, when parallel execution on the slow task is not possible, is to share processors among fast stages.

Let's call S_i at the process that addresses the stage number i of the pipeline. It receives either the input of the pipeline or the result of the previous stage P_{i-1} and produces its result, P_i, which is obtained after the stage's own processing and is delivered to the next stage for further processing, or is the result as the last stage of the pipeline that is stored, displayed on screen or

whatever.

We represent every stage with the following piece of pseudo-code:

```
process Si
        repeat
                receive subproblem Pi-1
                Pi = process(Pi-1)
                send subproblem Pi
        until !∃ Pi-1
        end
```

It is possible to think of the whole n-stages pipeline as a single algorithm like the following one:

```
process pipeline
        repeat
                receive subproblem P0
                P1 = process(P0)
                P2 = process(P1)
                ....
                ....
                Pn-3 = process(Pn-4)
                Pn-2 = process(Pn-3)
                Pn-1 = process(Pn-2)
                Pn  = processn(Pn-1)
        until !∃ P0
        end
```

or to think about it as a composition of functions in the following way:

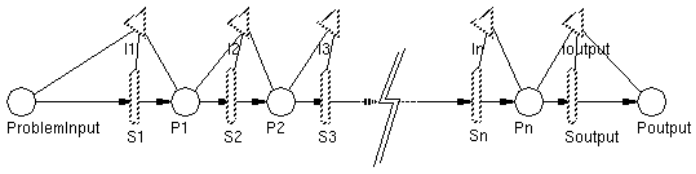
$$P_n = process_n(process_{n-1}(process_{n-2}(process_{n-3}(...(((process_2(process_1(P_0)))))...))))$$

where P_0 represents the input of the pipeline and P_n represents the output, each $process_m$ represents the processing at each stage. The intermediate variable assignments in the process and the functional composition in the functional representation represents the data exchange between the stages.
Using this approaches, we fail to represent aspects of the communication like bandwidth between adjacent processes, execution times, concurrency of multiple problems in the pipeline, etc. We only represent the resolution of a single problem using the pipeline, which is not enough.

System modeling

The simplest pipeline consists of only one stage but we will consider only pipelines with two or more stages, where parallel execution takes place.
As the data pipelining parallelism paradigm is based on the functional decomposition of the problem, the pipeline structure, number of stages, resolution time of each stage and other parameters are highly dependent on the particular problem that is being solved. There is no

individual Petri Net that can represent all data pipelines. The following network is a general representation of a data pipeline

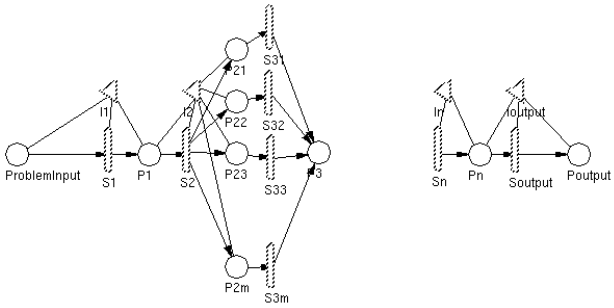


We see that this system receives tokens on `ProblemInput` place and moves them until they reach `Poutput`. The timed activities S_i represent the processing time at each stage. The input gates I_i governs the tokens movements along the pipeline, moving only one at a time.
The time distribution functions that govern each timed activity must be determined either empirically or from measures of prototypes from relevant stages. If the only estimation available for a stage is the expected mean execution time, we can use either deterministic time or exponential time distribution function. Using the exponential distribution function, we obtain a pessimistic approach to the execution time, due to its variance. On the other hand, using deterministic times we get optimistic execution times, because there is no CPU performance loss due to pipeline stalls caused by unexpected delays on particular stages.
If we assume deterministic execution times we can calculate the throughput of the pipeline. Lets call t_0, t_1, ..., t_n, t_{output} to the execution times of each stage, and lets suppose that we have a set of tokens in the place `ProblemInput`. Let us consider first a pipeline consisting of only two stages[19]. After a time t_0, a token is removed from `ProblemInput` place and moved to P_1. At that moment, two activities can be executed simultaneously: S_1 and S_2. Two execution times have to be considered now: t_0 and t_1. Lets suppose that t_1 is greater than t_0. A token can be moved from `ProblemInput` to P_1 at time t_0 representing that the task has `S1` been accomplished. At time $2.t_0$ a token could be removed from `ProblemInput` but as S_2 is not done yet, the token has to wait $t_1 - t_0$ until `S2` is done and P_1 is ready to accept a new task. If we suppose that t_0 is greater than t_1, a token is removed from `ProblemInput` at time t_0 and placed in P_1. Even though the token is removed from `P1` at time $t_0 + t_1$, the second stage, represented by will be idle until $2.t_0$ when the first stage finishes its part and starts again its processing. In both cases, the slowest stage slows down the throughput of the pipeline and introduces idle CPU cycles. We can see that in steady state, the system can only produce one result every $max(t_0, t_1)$. Repeating this reasoning, we can see that, for the general problem, the throughput will be lower bounded by $max(t_0, t_1, ..., t_n, t_{output})$, which means that we will not process faster than the slowest stage of the pipeline, the weakest link.
In many cases the inherent complexity of a single stage is very important and other actions have to be taken so as not to waste CPU power of other stages. Basically two approaches are taken: parallelize the slow stage or share CPUs on fast stages. We will analyze both options.

On the first case we have a stage whose resolution time is significantly longer than the rest and for some reason (i.e. real-time or simulation constrains) we need to speed it up. We are considering the case where money just cannot buy a faster CPU for that stage or the state of the art in microprocessors cannot solve the stage with a single CPU, no mater the chip

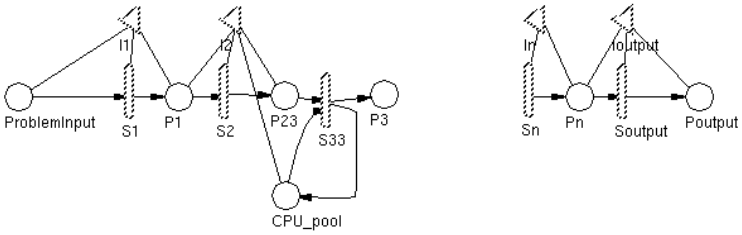
19 It is not difficult to see that with deterministic times, the throughput of a single stage pipeline is $\mathbf{1/t_0}$.

manufacturer we are considering.
What we need is the particular stage considered to be able to produce approximately the same number of results per unit of time than the other stages. The only option for tackling this problem is to use in parallel multiple CPUs within the stage. The particular way of parallelism chosen for the stage has to be considered for each particular problem, but if there is no correlation among the stages of the pipeline, it is possible to use a *master-slave* strategy for the stage, as there will be no communication between consecutive tasks[20]. The following Petri net represents a general pipeline with a *master-slave* parallelism on the second stage.



As it is done with the individual stages, the input gate controls the blocking of different stages, but it also controls the allocation of tasks to CPUs of the stage with multiple processors.
We are not speeding up the processing of each activity, but there is always a processor that can be allocated to a new incoming task and there is always a processor of the stage finishing with its task that can feed the next stage on the pipeline. If the processing takes equal times for the different input data possibilities, then we will even preserve the input order. It is very simple to achieve even more processing throughput simply allocating more processors to the stage. With this approach, we are not solving each input faster, but we are increasing the number of problems solved per unit of time.

It is possible to use an alternative representation for the previous case, in which we use multiple CPUs to increase the throughput of a stage. It is possible to model the stage with a pool of CPUs that are available for processing. Some modeling possibilities are lost, like CPUs with different numerical power, etc., but the system is adequate for equal processors.



The place `CPU_pool` was added to represent the set of processors that will address the paralleled stage. The input gate `I2` will check for the availability of CPUs or block until there is an available CPU, as it is done at all levels. As soon as the processing is done, the CPU is returned to the pool, making it available for reuse.
If there is some kind of correlation between consecutive stages in which the result of the stage

20 Not only *master-slave* can be used here, all other kinds of parallel classes can be applied.

depends not only on the current data set that is being solved but on the result of the previous one, it is not possible to use a straight master-slave approach; it is necessary to solve the data exchange between stages in some other way, maybe even using speculative parallelism or to address the problem really reducing the time of that stage.
It is possible to implement parallel execution of stages on more than one stage, and the parallelism used in each stage could be different stage to stage, leading to complex networks.

If the approach is not to make the slow stage faster, but to reduce the number of CPUs, it is possible to allocate multiple fast stages to only one processor. This would lead to sharing not only CPU cycles but memory, network bandwidth and other resources among the processes allocated simultaneously to one system. Depending on the set of processes joined many different kinds of interactions could happen. Depending on the length of the execution times, it is possible to simulate the concurrency of the processes (as it was done when modeling RC5) on the system or to simulate the real execution with prototypes so as to obtain good estimations of performance. With the previous data, we build a pipeline, like in the first case. Modeling the complexity of the pipeline plus the interaction within the shared system is theoretically possible, but numerically extremely intensive.

Performance results

We have already presented a method for modeling data pipelining systems. We still need to provide means that help deciding if the pipelined parallel execution vs. the single processor is convenient. The final goal of modeling is to determine as early as possible the best way to engineer a certain parallel system, how to design the pipeline, which way to partition the original problem into pieces, which of them to combine and which to separate into different stages, so as to reduce the complex set of different ways in which the system can be designed, or answer if current technology can address certain particular problem.
We will present now how to estimate the MES (Mean Execution Speed) and TET (Total Execution Time) based on the model described before.

Total Execution Time

Based on our system, we can calculate this measure as the time it takes the system to move all tokens out of the system. This implies that we have to simulate the whole resolution of the system. It would consist of placing as many tokens as necessary so as to represent the whole problem and let the system run until all tokens are moved from `ProblemInput` to `Poutput` place.
An alternative way of calculating the TET comes from network's steady state analysis. If we address the problem in this way, we can introduce a variation in the network's layout that avoids the absorbent configuration. We can make a cycle from the final state, the absorbent configuration where all tokens are placed at `Poutput`, to the initial one with all tokens placed at `ProblemInput` place, so after the processing is done, the network is restored to the initial configuration. Simply adding a timed activity that cycles all tokens to the initial configuration, we get a network that does not fall into an absorbent configuration. The process of restoring

the initial configuration must have associated timed transition with a known amount of time. We can simulate this new network in the steady state and measure the fraction of time that the network spends restoring the initial configuration. From that value we can then estimate the counterpart, that is the TET.
It is not always possible to calculate this measure due to complexity of the numerical solution. It can be the case that it might take too long to calculate the TET out of a complete execution simulation. If it was possible to estimate MES for the system, then it is possible to estimate the TET as the complexity of the problem divided by the MES, that is, how long it will take our system to consume all tokens at the processing average speed. On average, this estimation is adequate, as it was described before. For real time systems and highly complex pipelines that could consider pipeline halts, discarding tokens or other complex operations it is important to consider that it does not consider the behavior before and after the stationary phase. In the particular case of a pipeline, it is particularly important to have it running as long as possible on regimen state so as to take the better benefit of the execution. If the regimen state takes most of the execution time, then the estimation is adequate, otherwise, it has to be specifically considered.

Mean Execution Speed

If it is the case that we are calculating the MES after the calculation of the TET, then it is only an arithmetic operation remaining to compute MES according to the definition. On the other hand, if it is the case that computing the TET is not feasible, then it is possible to try estimating the MES before and afterwards, based on that value, to determine the TET.
As we discussed before, we need to estimate MES. We will estimate the regimen problem solving speed of our system (measured in work units per time units), that we call MES. This allows us to estimate the processing capacity of the system and also to estimate the time in which the stationary phase of the problem can be fulfilled.
The time in which the regimen phase is reached has to be calculated in some way that is problem dependent. After the system is in its regimen phase, one of several standard techniques (regenerative simulation, batch means, etc.) can be applied to determine the MES. Changes in the network can be done to simplify the determination of the steady state like adding infinite initial tokens or making cycles so as to keep the overall number of tokens constant according to the time evolution. The last alternative is generally preferable for the sake numerical simplicity.
If the system was modeled according to the previous recommendations, there has to be a `Probleminput` place which holds the tokens that model the problem space, and a `Poutput`, that models the pieces of the problem leaving the pipeline.
Even though we could theoretically analyze the processing speed based on an infinite set of tokens in the `Probleminput` place, it is numerically simpler in our Petri Net to model a finite number of tokens cycling through all the stages. To achieve this, it is useful to add an instant transition from `Poutput` place to `Probleminput` place. Keeping the number of tokens constant, we can estimate how much time it takes a token to do a cycle after the steady state is reached. This time is the MES.
Each cycle of a token represents the completion of a work-unit, that means, if we are able to count the number of cycles that all tokens perform within a certain period of time, then we know the number of work units that can be solved in that period of time. We can count the number of cycles associating signals to a specific place (i.e `Poutput`). Afterwards, using the

batch means method we can estimate the MES.
It is also possible to perform the same study that was depicted when analyzing the TET with a timed activity instead of an instant one and indirectly determining the MES.
As a rule of thumb, it is important that on the steady state there is always more than one token on the `Probleminput` place. If all tokens are consumed, it is possible that a stage is willing to process, but there is nothing to process. If there are always tokens on the `Probleminput` place, it means that there is always more work to be done than the pipeline can process, and thus, there is no idle stage due to token shortage. In that situation, we are solving the problem as fast as we can.
Another simpler way of estimating the MES while in regimen state is based on the slowest stage. As it was seen before, the throughput of the pipeline is one of its slowest stage, and hence, its MES. This measure is even a rougher estimation, but is useful when facing complex pipelines as it gives a simple to calculate and at-hand estimation.

4.5 - *Divide & Conquer*

This approach is widely known in sequential algorithm development: a problem is divided into two or more subproblems, each solved independently and their results are combined to give the final result. In most cases, the subproblems are just smaller instances of the original problem (and can be solved using the same algorithm, working on a smaller set of data). This gives leads to recursive solutions implemented with stack structures for recording execution evolution and invocations, etc.. In parallel *divide and conquer*, the subproblems can be solved at the same time, given sufficient parallelism. Because the problems are independent, no communication is necessary between processes working on different problems.
There are three generic operations: *splitting*, *computing* and *joining*, which are organized particularly on each algorithm. The general structure is that, first of all, each algorithm receives a piece of work to be solved. It does some processing so as to determine if it is going to address the resolution of that piece of work by its own or if it is going to spawn child processes. Before spawning, splitting takes place and the sub-problems that will be allocated to children are created with some particular problem dependent criteria. If child processes are spawned, then before processing goes on, it is necessary to wait for children to finish processing and return their results. Joining the children results and producing the final result of the stage follows. If spawning did not take place, local processing would take place and the final result would be the one locally obtained.
The execution of *divide and conquer* algorithms leads to tree-like structures (in many cases binary trees). The following algorithm represents a single node process, that, depending on the input may act either as an inner node or a leaf node:

```
process divide-and-conquer
      repeat
            receive subproblem P0
            if condition(P0) then
                  R0 = process(P0)
                  send(R0)
            else
```

```
                    (P0i, P02, .... , P0n) = split(P0)

                    for i in 1..n
                          send(P0i)

                    for i in 1..n
                          R0i = async receive(P0i)

                    R0 = join(R01, R02, .... ,R0n)
             endif
      until !∃ P0
      end
```

The resulting shape of the tree and structure of the resulting execution depends not only on the problem, but also on the data set that the algorithm receives. There is no single scheme representation for the class of divide and conquer parallel applications that covers the general case. As we can not depict a single network that describes this problem, we will determine how to build one. Different algorithms may vary on the number of nodes that the split process produces and may even introduce extra communication to the vertical one established by the split-compute-join sequence, leading to graphs in which communication pattern may become more complex. On the other hand, the same algorithm with a different input may lead to significant differences in size, which involve detailed considerations related to system resources.
As a consequence, it is not always possible to determine exact shape and size of a particular divide and conquer algorithm until de data set and the algorithm are known.
Another factor that is normally controlled is the spawning of tasks. The number of tasks on a divide and conquer strategy grows exponentially on the depth of the tree, leading to extremely fast exhausting of resources if the spawning of new tasks is not under control. In most of the cases the metrics used for splitting take into consideration the number of nodes the cluster has, so as not to outnumber the processors with tasks. If we build a system with *n* processors, the general rule is not to have more than *n* processors doing heavy computation. Different strategies can be used to limit the number of processes running. If the algorithm we are building spawns *p* children at each level, processing is only done at the leaf nodes and we have *m* processors, then we can limit the depth of our tree to $\log_p(m)$ if we want to have all processing nodes running at once. If this is not possible due to memory or other constrains, then it is possible to control the number of concurrent running processes and use some strategy like DFS to traverse the resolution tree.

System modeling

When we want to represent this system on a Petri Net, we find that it is not possible to sketch a net in which we have a pool of CPUs and a set of tasks to be accomplished because we lose the interaction and interleaving of processing and sharing information, as we found out when discussing SPMD class. We present here a procedure for constructing the Petri Net associated to a given *divide and conquer* algorithm that executes simultaneously all the processing nodes on separate processors. We will refine this algorithm further so as to consider the execution under spawning controlled conditions.

We will introduce first the definition of the places and transitions that will conform our net. Each process will cycle once through the sequence split, compute and join, and will be represented by three places. We do not model re-use of places like re-use of processes because the most general divide and conquer situation comprises process disposal after its cycle. We will call S_{ij} the place that represents a process that is in its splitting stage, and is the number *j* on the i^{th} level. We will call SP_{ij} the timed activity that represents the time spent during the splitting process.
After a certain amount of splitting, the problem is sufficiently reduced and computing can take place. At the last level, the processing is represented with places labeled C_j and timed activities labeled Cmp_{ij}. For each place Sij representing the splitting there is a J_{ij} place representing the joining that happens after the children have ended up and the stage's recursion ends.
The processes will either be represented by a Sij - Jij couple if it is an inner node or by C_j if it is a leaf node.

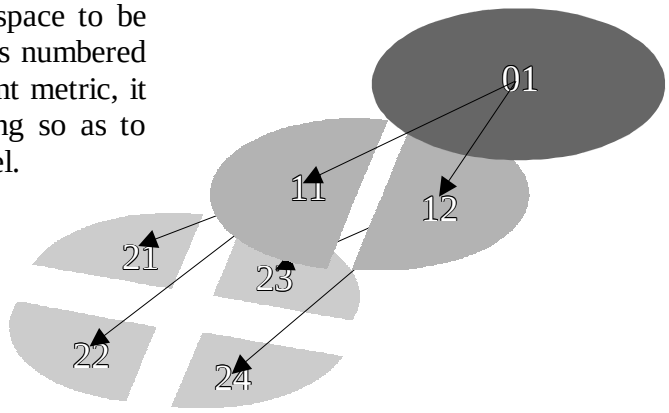
Lets call *Ch* to the number of child that a node can spawn in the recursion

for *i* in [1..(depth-1)]
  for *j* in [1..Ch^{i-1}]
    add a place labeled $S_{i-1,j}$
    add a timed transition labeled $SP_{i-1,j}$
    add an input gate $IS_{i-1,j}$
    add a place labeled $J_{i-1,j}$
    add a timed transition labeled $JP_{i-1,j}$
    add an input gate $IJ_{i-1,j}$
    add an arc from $S_{i-1,j}$ to $SP_{i-1,j}$
    add an arc from $JP_{i-1,j}$ to $J_{i-1,j}$
    add an arc from $Is_{i-1,j}$ to $S_{i-1,j}$
    add an arc from $J_{i-1,j}$ to $IJ_{i-1,j}$
    add an arc from $Is_{i-1,j}$ to $SP_{i-1,j}$
    add an arc from $IJ_{i-1,j}$ to $JP_{i-1,j}$
    if i > 1
      add an arc from $S_{i-1,j}$ to $IS_{i-2,\lceil j/Ch \rceil}$
      add an arc from $SP_{i-2,\lceil j/Ch \rceil}$ to $S_{i-1,j}$
      add an arc from $J_{i-1,j}$ to $IJ_{i-2,\lceil j/Ch \rceil}$
      add an arc from $J_{i-1,j}$ to $JP_{i-2,\lceil j/Ch \rceil}$

```
for j in [1..Ch^depth]
      add a place labeled C_j
      add an arc from  SP_depth-1,⌈j/Ch⌉ to C,j
      add an arc from  C_j to JP_depth-1,⌈j/Ch⌉
      add an arc from  C_j to IS_depth-1,⌈j/Ch⌉
      add an arc from  C,j to IJ_depth-1,⌈j/Ch⌉
```

Some parameters of the network still need to be defined. To have a fully defined network, we still need to determine the distribution functions for the timed transitions and the number of tokens in the initial configuration. Before going on to the definition of the network, we need to state something about its complexity. The complexity of the network can grow considerably. For each process[21] two places, two transitions, one input gate and six arcs are added. The resolution of the resulting network can consume some CPU power and could take significant effort.

Lets apply the procedure to a simple configuration. Our example configuration consists of a space that is going to be solved using a divide and conquer approach. The depth of the tree will be three levels and each space division will partition the space into two similar subspaces, thus there will be four computing nodes and three join-split processes, leading to six places. The following figure represents the successive division of the space until the splitting reaches a level that makes it addressable for a single process.



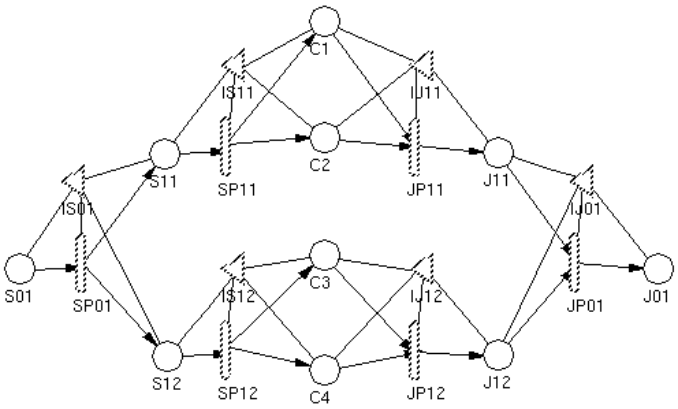
The first oval represents the problem-space to be solved, that is addressed by the process numbered “01". Applying some problem dependent metric, it determines that there has to be splitting so as to partition the space and solve it in parallel.

At the next level, the problem is partitioned in two subproblems, numbered “11” and “12”. Each of this processes are again too big to be addressed by a single processor, and thus, they execute the partition step, splitting themselves into two subproblems each.

At the next level, the subproblems “21”, “22”, “23” and “24” are small enough, according to the metric, to be addressed by a single processor, so they are solved in parallel. After the computation finishes, the results are returned and the hierarchy is traversed upwards, joining branches, until the final solution is found.

Applying the procedure, we get the following network

21 except C_j ones.



What we have now is a picture of inter-process communication schema. As we want to represent the problem resolution, we need to represent the problem evolution through the network. We have to choose a representation for the problem space. The complexity of the network can grow significantly as the number of nodes grow. Current Beowulf projects face hundreds or even thousands of nodes that can lead to extremely large nets.

What we did with the previous models was to determine a certain adequate processing unit in which we partitioned the problem, determined the expected time for a certain system to process that unit, represented by a token. Making those tokens run through the network we model the problem resolution. We choose tokens that represent the same portion of the problem throughout the network.
Following that kind of reasoning we can partition the problem into units that represent the fraction of the problem that would be addressable by a single process, a C_i place. The problem would be then partitioned into Ch^{depth} pieces. All the tokens would be initially placed in the place named S_{01} and the simulation ends when all tokens reach J_{01} place, representing that all the partial solutions finally were joined back into the initial, 01, process.
The intermediate timed activities control the movement of tokens and synchronization. We have to use them because we need to alter the behavior of the standard Petri net. In our case, when tokens are moved, they are all moved at once, that is, when the problem is partitioned into *n* pieces, each piece (consisting of one *n-th* of the tokens) is moved "as one" to the child process that is going to solve it. If we use plain Petri nets, each token is moved independently of the rest, losing the meaning of the partitioning we intend, and increasing also the number of states to be considered in the resolution. If we allow individual token moving, we will maintain the proper semantic. All the input gates are introduced to preserve this: IS_{xx} input gates control the partitioning associated with SP_{xx} activities while IJ_{yy} input gates control the joining associated with JP_{yy} activities.
SP_{xx} activities will split the problem, generally in equally sized pieces. If the size of the problem is a factor that influences the communication time between S places, then the activity SP_{xx} can use the number of tokens on place SP_{xx} as an input that determines the time spent on that activity.
$JP_{depth-1,y}$ activities represents the processing done on processing nodes. We are representing the time spent on two processors with only one activity instead of one timed activity per processor. If the problem is equally distributed among processors and they have similar processing power, then the modeled system will behave as the real system.

The tokens that moves through J places represent solved parts of the problem that are joined together. The timed activities represent the time spent joining and the communication among the different processes, until the tokens reach the J_{01} place. Once again, the input gates are used to alter the behavior of a standard Petri net and move sets of tokens at once.
In the case that the joining stage simply moves back single tokens, representing maybe single solution values, the input gates could be removed, leaving the standard Petri net behavior. Whenever this is possible, it should be done, so as to simplify the resulting network.

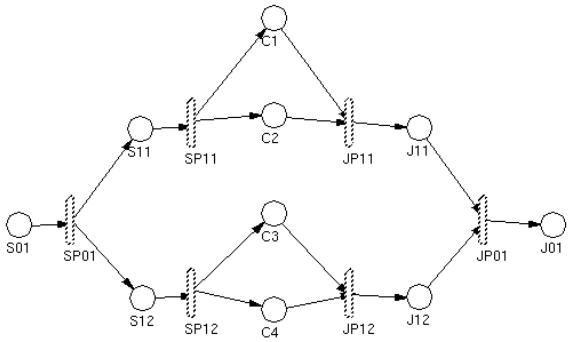
There is another way to represent this, but it does not correspond to the way we have been representing the problem within the system. It is possible to use a single token moving from stage to stage that represents different parts of the problem on different places. Up to now we have been working basing our modeling of the problem with tokens that represent certain "work units" that are solved as they are moved through the network. In a way we are giving an invariant value to the token all through the network. The problem evolution is represented with the consumption of this tokens, until there is no token left. The timed activities here that represent the computation are proportional to the CPU power of the system being modeled.
Using the alternative approach described, we can think of tokens as marks that represent only completion of stages, but have no relation with portions of the problem independent from the position they occupy. One consequence is that the timed activities can not be uniform in relation to tokens. Depending on the place of the network, they are placed, the completion of each task could take different amounts of time, leading to higher complexity in the calculation of the distribution functions for each activity.
Another problem is that the system behavior has to be well known so as to write a static network to simulate the resolution of the problem. It is necessary that the process intercommunication and solution evolution is known before the model could be written. As this is the most general case with divide and conquer parallel systems, it is possible to model them with this approach.
The resulting network is simpler from the point of view of number of arcs and also input gates, as they disappear. The computing resources needed to solve them also decrease, as the number of states of the network falls dramatically.
From the point of view of the tuning and adjustment of the parameters it might be more difficult to calculate all parameters associated to each process at each level/place.
The following procedure can be used to create a network that represents the resolution of a divide and conquer parallel problem using the previous approach:

Lets call Ch to the number of children that a node can spawn in the recursion

for i in [1..(depth-1)]
  for j in [1..Ch^{i-1}]
    add a place labeled $S_{i-1,j}$
    add a timed transition labeled $SP_{i-1,j}$
    add a place labeled $J_{i-1,j}$
    add a timed transition labeled $JP_{i-1,j}$
    add an arc from $S_{i-1,j}$ to $SP_{i-1,j}$
    add an arc from $JP_{i-1,j}$ to $J_{i-1,j}$
    if i > 1
      add an arc from $SP_{i-2,\lfloor j/Ch \rfloor}$ to $S_{i-1,j}$
      add an arc from $J_{i-1,j}$ to $JP_{i-2,\lfloor j/Ch \rfloor}$
for j in [1..Ch^{depth}]
  add a place labeled C_j
  add an arc from $SP_{depth-1,\lfloor j/Ch \rfloor}$ to $C_{,j}$

add an arc from C_j to $JP_{depth-1,\lfloor j/Ch \rfloor}$

The previous procedure can be seen as the first one presented for this model with all the steps regarding to input gates deleted.
The resulting network of applying the procedure to our example follows:



As we discussed before, a single token represents the whole problem and should be initially located at place S_{01}. SP_{xx} activities represent splitting and communication time as the token moves processes while partitioning. The number of tokens grow in this network, and can reach a maximum of Ch^{depth} when all tokens are in C_j places.
$JP_{depth-1,y}$ activities represent the processing done in processing nodes. We are representing the time spent on two processors with only one activity instead of one timed activity per processor, as we did in the previous model. If the problem is equally distributed among processors and they have similar processing power, then the modeled system will behave as the real system. The $JP_{depth-1,y}$ activities definition is the same in this model and in the previous one for divide and conquer parallel algorithms.
The tokens that move through J places represent solved parts of the problem that are joined together. The timed activities represent the time spent joining and the communication among the different processes, until the tokens reach the J_{01} place, when all the splitting, computing and joining cycle is finished.

Performance results

We have already presented a method for modeling divide and conquer systems. We still need to provide means that help deciding if the parallel execution vs. the single processor is convenient. We will present now why it is not possible to estimate the MES (Mean Execution Speed) out of the Petri Net and how to estimate TET (Total Execution Time) based on the model described before.

Total Execution Time

The total execution time of the problem is the time from the initialization phase until the

process reaches its ending. This is a transient measure, which is particularly important in real-time or quasi real-time parallel systems, but it is generally important on every system that we code: we would like to know how long it will take to find a solution to our problem.
Based on our system, we can calculate this measure as the time it takes the system to move all tokens out of the system, from S_{01} to J_{01}. This implies that we have to simulate the whole resolution of the system, that means, we have to model the whole system as described before, place a number of tokens in the initial place which represent the problem in our selected work unit size and run a simulation until all tokens are removed away from the system.
Not all simulation tools allow the determination of the time elapsed to reach a specific state. If the tool that is being used allows us to calculate network's steady state, we can introduce a variation in the network's layout. We can make a cycle from the final state, the absorbent configuration, to the initial one, so after the processing is done, the network is restored to the initial configuration. No matter which of the suggested work decomposition schemes was used the cycle would consist of a timed activity that moves tokens from the place J_{01} to S_{01}. The number of tokens moved depends on the semantics given to the tokens according to which scheme was used. In the first case, all tokens that model the problem must be placed back in the initial state, while in the second scheme, only one token ought to be placed in the place S_{01}. The process of restoring the initial configuration must have an associated timed transition with a known amount of time. We can simulate this new network in the steady state and measure the fraction of time that the network spends restoring the initial configuration. From that value we can then estimate the counterpart, that is the TET.
It is not always possible to ascertain this measure basing our forecast on simulation due to the complexity of the numerical solution. It can be the case that it might take too long to calculate the TET out of a complete execution simulation. An obvious lower bound for this system could be estimated multiplying the total number of work units to be solved by the average time it takes for a processor to solve that work unit divided by the number of processors addressing the problem. This is a far too rough estimation that could even present errors of orders of magnitude with a finer lower bound. If no other lower bound is available, it could be used with extreme care, but is not completely meaningful. It obviates all communication times, network and other devices latency, all system overhead, all partitioning and joining times, etc. It only gives what is related to processor performance, but it does not take into consideration the rest of the parallel system components.

Mean Execution Speed

If we are calculating the MES after the calculation of the TET, then it is only an arithmetic operation remaining to compute MES, according to its definition.
As it was said before, we need systems that present a stationary behavior so as to estimate the average number of pieces of work that are solved in a unit of time. According to the way we modeled the system, there is no such thing as a regimen behavior that can be studied. The evolution of the system implies tokens passing through the network dynamically, without any token feedback or loop that can be used to get an average measure. Furthermore, the network we presented has only one set of processes that perform the resolution, whilst the rest do the partitioning and the joining.
Even though we could not determine the MES out of the net, we can theoretically analyze the processing speed based on the processing capability of the processing nodes, which was known when we calculated the rate of $JP_{x,y}$ activities, and the total number of tokens at the C nodes,

where processing takes place. The average number of nodes processed at that stage is what we can call MES out of this system. The main problem is that we are obviating the partitioning and joining times. As we said before, we are able to perform this simplifications where $T_r >> T_i$ and $T_r >> T_e$, thus, it is valid that $TET \simeq T_r$. In this kind of parallel systems this is not valid on most cases and we can conclude that estimating the TET out of a MES like the one described before could be done on a small number of particular problems.

4.6 - *Speculative Parallelism*

This approach is used when the previous parallel models are extremely hard to use or implement. The situation arises either due to complexity of the data interdependence among different processors or when unpredictable and diverse times of tasks completion generates excessive execution processors stalls and forces the parallelism system to assume most probable counterpart result to follow its calculation. If the optimistic execution results is confirmed, current state is check-pointed and execution continues. If the optimistic result assumed was assumed wrong, then the current state of the system is rolled back to the previous check-pointed state and execution is resumed from there, but following the right execution path. In some asynchronous problems like discrete-event simulation, the system will attempt the look-ahead execution of related activities in an optimistic assumption that such concurrent executions do not violate the consistency of the problem execution.
Another possible use of this scheme is to address a problem with different algorithms, generally, not deterministic ones or a mix of deterministic and simulated ones. Whenever a solution (or an appropriate estimation) is found, the rest of processors are stopped, the solution is shared and they follow up from there on. We can exploit the benefits of many algorithms this way. It is very easy to use this technique to speed up simulated-annealing, Monte Carlo, Tabou search and GRASP simulations just choosing proper random number generators for each system.
According to the way we are modeling the systems, we do not model stages on the process resolution but amount of work remaining. It could have been possible to use colored tokens or other Petri net extension to make differences on the tokens that could both differentiate them and put extra semantics there. For our purposes there was no need to take that approach so as to represent the system evolution. In this case, when a roll-back situation needs to be modeled, it is quite intuitive at first sight to think of colored tokens to represent the regression to a previous state, but the method we used in dealing with the state regression is to put more tokens on the place that represents a process's remaining work when another process, associated to it, violates a constrain and regression occurs. We do not model which part of the work has to be done again, but we represent the amount of work added after a rollback. All we need to know is how often it happens and the average amount of work rolled-back.

Lets assume that we have a function V which returns the set of neighbors of a given processor P_i. Using these functions, we know which processes to signal when we solve a part of the problem. Knowing our timestamp[22] they can decide their execution violates any constraint or

22Not only time evolution can be used for synchronization. We base our analysis on this figure knowing that

not so as to rollback or continue. In our case, we will randomly decide if rollback occurs or not because we can not model an algorithm that we do not know precisely. This random generation of rollbacks has to be controlled carefully as is an important source of error: if it is too often, the system will be doing little; if it is to seldom, we will model a system that outperforms the real one.
Each process itself executes basically a simple sequence of stages, represented by the next piece of pseudo-code:

```
process speculative-parallelism, Pi
      thread0
            repeat
                  retrieve subproblem P0
                  R0 = process(P0)
                  Update local simulation time
                  for Pj in V(Pi)
                        send(Pj,R0,timestamp)
            until !∃ P0
            end
      thread1
            repeat
                  receive(V(Pj),Rx,timestamp)
                  if check(P0,Rx,timestamp)
                        rollback(P0,Rx,timestamp)
            forever
            end
```

The execution is represented as two concurrent threads, one responsible for the execution itself and the second one is listening to adjacent processes results checking for violations of the constraints.

System modeling

We want to represent a general speculative parallelism problem using Petri nets. We found it is not possible to use a specific Petri net to represent all cases because an important part of the information would be lost, specifically process interlocking, communication, splitting, etc. We will present a procedure that produces a Petri net that models a given specific problem.
Something has to be said regarding the execution and problem representation. Whenever we are using heuristics or simply algorithms whose execution time can not be estimated as a function that depends on processing speed and problem size, we are facing a situation in which we can not specify clearly the problem size. Lets say that we are modeling an optimization algorithm that is going to use a GRASP heuristic and we want a solution with a certain level of quality. We cannot state how long it will take the algorithm to reach the level of quality expected. As we cannot control the random component in this solutions, we can not assert if the solution is going to be obtained within a certain number of operations or within a specified period of time. It is possible that we find not only a valid solution but the optimal solution with our first heuristic execution, and it is also possible that after an arbitrarily long period of time, no acceptable solution is found.
When we model this problems we have to have a certain level of confidence in our heuristics,

other parameters can be used.

and we must have an estimation of the number of experiments we should run to obtain a solution. Maybe running some more experiments we get a better solution, but we must be confident that after a certain number of experiments it is most probable that we have a solution. With this assumption we now have a problem that we can measure and represent. We can estimate number of experiments, estimated time for each, etc. Problems like SETI@home[23] (even though they are not using heuristics for each experiments) rely on many random factors like the existence of extraterrestrial intelligence, etc., so the problem size cannot be determined. What can be determined is the amount of data gathered by the telescope daily, but it is not possible to determine how much data and processing is going to be needed for the problem resolution: we do not even know if there is a solution. What we can model is the speed at which daily information can be processed, etc. We can estimate MES but not TET in this situation.

In our study of *speculative parallelism* analysis, what we will model is the resolution of an amount of work that we believe will be enough for us to get a solution. With that concrete problem size estimation we will proceed with problem partitioning in tokens as we do with all problem classes.

So as to model this system, we will present firstly the definition of the places and transitions that will conform our net. Each process will cycle on its main loop solving pieces of work until there is no more work to be done. We will model the `thread0` of each process as a place and a timed activity that removes tokens from the place, representing the evolution in the process resolution. `thread1` will be modeled as a place, fed by processes neighbors with an instant transition associated which will model the checking for the need of rolling back, in our case, choosing randomly if rollback would happen.

Lets call P_i the place where the tokens representing the work remaining for process *i* would be placed, and lets call W_i the timed activity that removes tokens from P_i as they are completed and the associated processed are signaled. The signals arriving from neighbors reaches a place S_i, where an instant transition, R_i determines if rollback occurs or not. We also have a function *V* that returns the set of indexes of the adjacent processes to a given one.

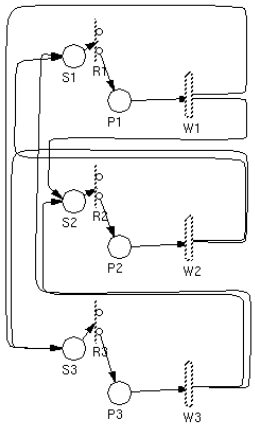
```
Let m be the number of processes

    for i in [1..m]
       add a place labeled Pi
       add a place labeled Si
       add an instant transition labeled Ri with two cases
       add a timed transition labeled Wi
       add an arc from Pi to Wi
       add an arc from Si to Ri
       add an arc from the second case of Ri to Pi
       for J in V(Pi)
                add an arc from Wi to SJ
```

Some parameters of the network still need to be defined so as to have a fully defined network. We still need to determine the distribution functions for the timed transitions and the number of tokens. The complexity of the network will be much more controlled than in previous cases. The resolution of the resulting network will not consume as much CPU power and time as other methods, but the drawback is that due to the situations this method is applied, not much

23 It must be kept in mind that SETI@home does not fall within *Speculative Parallelism* class and we are only using it as an example of an unbounded problem.

level of detail can be achieved.
To complete the network definition we need an appropriate distribution function that models the expected time spent by each processor solving every time-step and the distribution function that models if a solution found by a neighbor is better than his own. Generally, this second estimation will be assumed as uniformly distributed for the neighbors. With this functions estimated for all neighbors the net is fully defined.
Lets apply the previous procedure to a set of three machines solving a problem performing a discrete-event simulation. Lets suppose that the system designers are planing to simulate the time evolution using three systems: one of them running a Monte Carlo simulation, another simulated annealing the problem and the last of them using batch means. Each of them uses its own method for simulating each time-step. They try to address the problem, finding a solution that lies below a certain level of acceptable error. All problems will find their solution at a different pace, as they are addressing them on different ways. The overall solution will pick the best solution found by each processor each timestep and will make them go from there on. The optimistic presumption that each process will assume is that its solution is the best and they will continue with their solution lest they get a better one from other. The rollback means discarding their findings and restart from the best time and solution given by a neighbor. The amount of work to be accomplished by every process is represented by tokens, each of them plays the part of a time-step and the number of initial tokens is calculated dividing the simulation time by the time-step time. After each process finds his solution, it shares its findings with the two remaining processes. All processes have the same average speed to solve each time-step. Assuming that each of them have the same probability to find the best solution, the probability that there occurs a rollback is of two thirds and the average rollback is of one token back.

The following Petri net represents the system:



Performance results

We have already presented a method for modeling systems that use parallelism for their resolution using the speculative parallelism paradigm. We still need to provide means that help

deciding if it is convenient the parallel execution vs. the single processor one. We will present now how to estimate the TET (Total Execution Time) and the MES (Mean Execution Speed) based on the model described before.

Total Execution Time

The total execution time in this kind of problems may vary something from the general definition, as we are not sure that within that time we will certainly solve the problem. In this case, the total execution time is the time from the initialization phase until the process performs all the work that we are confident is needed to find an acceptable solution, even if the actual solution is not found.
Based on our system, we can calculate this measure as the time it takes the system to move all tokens away from the system, out of P_i places. This way of calculating it implies that we have to simulate the whole resolution of the system.
If the tool that is being used allows us to calculate network's steady state, we can introduce a variation in the network's layout. We can make a cycle from the final state, the absorbent configuration, to the initial one, so after the processing is done, the network is restored to the initial configuration. We can introduce an activity with an associated input gate that monitors activity in the network. If all processing is done, that is, no token remains in the network, the initial configuration is restored. The process of restoring the initial configuration must have an associated timed transition with a known amount of time. We can simulate this new network in the steady state and measure the fraction of time that the network spends restoring the initial configuration. From that value we can then estimate the counterpart, that is the TET.
If we have already estimated the MES for that system, it is possible to estimate the TET as the complexity of the problem divided by the MES, that is, how long it will take to our system to consume all tokens at the processing average speed. On average, this estimation is adequate, but it does not consider the behavior before and after the regimen phase.
An important source of error to this estimation is due to the time the system runs out of the steady state. In general, the MES will soothe this effect because it will correspond not only to an average of multiple run but an average of different algorithms.

Mean Execution Speed

If it is the case that we are calculating the MES after the calculation of the TET, then it is only an arithmetic operation remaining to compute MES according to the definition. On the other hand, if it is the case that computing the TET is not feasible, then it is possible to try estimating the MES before and afterwards, based on that value, to determine the TET.
As it was said before, try to estimate the average number of pieces of work that are solved in a unit of time, and that is what we call MES. This allows us to estimate the processing capacity of the system and also to estimate the time in which the stationary phase of the problem can be fulfilled.
The time in which the stationary phase is reached has to be calculated in some way that is problem dependent, but in general, successive terminating simulations can be run until the terminating state falls within the boundaries of the regimen state. In most cases, the regimen state is reached in short periods of time due to the loose interleave of processes.

Even though we could theoretically analyze the processing speed based on an infinite set of tokens in the P_i place, it is numerically simpler in our Petri Net to model a single token cycling through all the stages. If no rolling back occurs, the speed is one of the fastest processes, and no simulation would be needed. This particular calculation is a simple one and is a lower bound for the MES.
The need for simulation arises due to rollbacks. It is useful to make a small modification on the net so as to estimate the number of rollbacks. Lets add a couple of places Rol_i and $Done_i$ for each process, one arc from W_i to $Done_i$ and another arc from the second case of R_i to Rol_i.
Performing transient simulations is is possible to count the number of tokens collected on the places Rol_i and $Done_i$ for each process, representing the number of rollbacks and solved time-steps respectively. The difference between Donei and Roli is in successive time intervals is the MES for each process or MES_i. The systems MES is calculated from the individual ones according to the particular relation of the processes, but weighting MES_i with processing speed and number of rollbacks of each processor.

4.7 - *Hybrid models*

This approach is taken when real applications do not lie exactly within the definition of the previous groups or, in some cases, it is useful to mix different elements of the different paradigms. They are not generally found on small applications, but in situations where it makes sense to mix them in different parts of the same program.
The way this systems are modeled consists of isolating the different conceptual models, modeling them according to their corresponding models. The partial models are coupled back together completing the whole system. If the modeling is done using UltraSAN, then the individual models can be joined using the composed models. This is useful to keep the individual networks corresponding to each model separated from the whole, keeping them simpler and conceptually properly corresponding to their identified stages.

5 - Case Studies

5.1 - Introduction

The objective of this chapter is to illustrate the usage of the previous models on real parallel applications. We will compare predicted performance estimation of the models with actual system performance. We will choose some parallel applications arbitrarily, we will fit them within the proper class of parallel application, we will apply the corresponding procedure to obtain a Petri Net that models it and then we will estimate the performance of the system analytically and contrast the estimation with ground measures from real systems running the applications we picked.
The objective of these studies is not to develop benchmarking or cluster loading tools or industrial parallel applications. We try to explore some cluster performance aspects, isolate them and apply our theoretical analysis. We need simple, understandable and predictable problems that can be addressed easily under different conditions like number of CPUs, etc. The final goal is to code simple parallel algorithms (accessible and easily comprehensible while parallel) that would help to understand complex interactions of the system performance.
The first experiment will consist of a domain-decomposition application that will perform operations over a matrix. The second experiment will consist on the heuristic resolution of a np-complex problem using a metaheuristic.
At the end of this chapter we present the results of a small test performed to overload a system with an excessive number of tasks that allows us to understand the cost of assigning more than one CPU bound task to a system. This study will be presented as an annex because no modeling or parallel execution was performed.
It can be seen that the selected experiments are very different one from the other. The first one is *classically* coded on C + PVM and run on Linux (even though not on a Beowulf cluster). The second experiment exploits parallelism through parallel threads invoking remote objects through RMI in JAVA. We shall show that the models introduced work in both scenarios. This is important, as the model templates are not tied to particular hardware, software or algorithm configurations and can be used in many heterogeneous situations.

5.2 - SPMD example application: *Mat*

This application solves the general case of a time series of matrixes of dimensions $m \times n$ in which any point can be calculated as a function of itself and its immediate adjacent in the previous point in time. Lets represent a point with coordinates *i,j* at time t_0 with $x(i,j)_{t_0}$.
The *mat* program can be used to solve problems where the following equation holds:

$$x(i,j)_{t_{n+1}} = f(x(i,j)_{t_n}, x(i-1,j-1)_{t_n}, x(i-1,j)_{t_n}, x(i-1,j+1)_{t_n}, x(i,j-1)_{t_n}, \ldots, x(i+1,j+1)_{t_n})$$

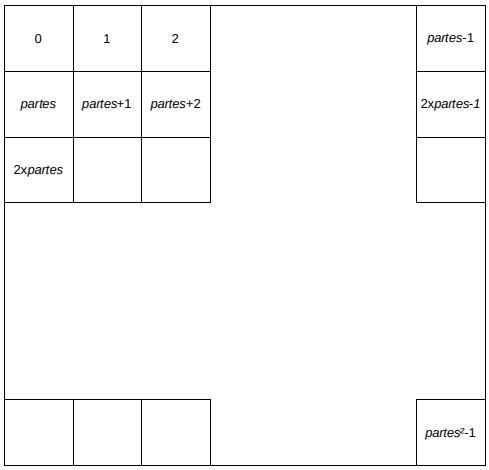
The implementation considers the wrapping of the matrix, both horizontally and vertically that permits simpler coding for topographically closed scenarios mapped to matrixes (ie. Geoides bodies to matrixes for weather analysis). The border conditions or wrapping is determined by the *f* function, that determines what to do with wrapped neighbors. We can rewrite previous equation to consider wrapping the following way:

$$x(i,j)_{t_{n+1}} = f(x(i,j)_{t_n}, x(mod_m(i-1), mod_n(j-1))_{t_n}, x(mod_m(i-1),j)_{t_n}, x(mod_m(i-1), mod_n(j$$

It is possible to generalize this problem to more complex scenarios considering more intricate patterns of neighbors and also more points in time, but current *mat* program suffices for a relatively interesting set of problems, and gives us an understandable and simple framework for analyzing the parallel execution of this set of problems. This application is also interesting from the point of view of the CPU load it obtains on the system. The complexity can grow as much as we need with pretty small matrixes. The program is written in C and can be linked to any object that exports a function called `doprocess` that implements the time evolution function *f*. The library used to perform the interprocess communications is PVM.

The process can be invoked both through the command line and a PVM console. It requires two parameters and a third one is optional. The first parameter, ***partes***, is a number and it counts the number of parts (horizontal and vertical) in which the input matrix is going to be splitted. The problem would be solved by ***partes***^{2} concurrent cooperative processes. The second parameter, ***cant*** is the amount of time steps that the parallel system will be run. The last and optional parameter is the pathname to the data file. If omitted, the data will be read from a file called `datos` in the execution directory. This parameter could become important if the initial data set is big. The data could be copied to machine local directories so there is no overload on a file server at start up. The output, a matrix with the dimensions of the input one, then is written to the file `/tmp/salida` in the machine where the parent process runs.

The program has an initialization stage in which the initial process, the father, generates ***partes***^{2} child processes, assign them a position inside the main matrix ($m \times n$) and communicates the *PIDs* of their neighbors to all of them. Each portion of the matrix is assigned a number starting from 0 on the upper left corner and increasing by one from left to right, top down according to the schema. The numbers are Real numbers represented internally as double precision (64bit) float.
From the general formula we have presented, we can determine the pattern of communication that will occur. The function uses at most all adjacent points in the matrix to a certain one to calculate the next value of that point. Whenever a child process is calculating a point inside the matrix, it always has all the necessary information so as to finish the calculation, but when we are considering a point in the border, then the information necessary can be shared with up to three other neighbors when we consider the information needed to calculate the elements in the corners of the matrix. We will analyze deeper this point later.



After the initialization stage finishes, the system loops ***cant*** times performing this simple sequence of steps: *transmit*, *receive*, *process*. We organized the communication pattern relying on the semantics of PVM. We used non-blocking *send* primitives and blocking *receive* primitives. This allowed us to code the processes so that they can share results with neighbors regardless of their situation: the messages will wait in communication buffers until the receiving process needs them, without blocking the sender; the sender will block himself until he has all the information needed to do his computation thoroughly. When the process tries to fetch the information needed it checks orderly the messages from neighbors. If a message is already there it is processed, but if the expected message is not there, the process gets blocked until the message comes. At that point he has already sent all the information needed by their neighbors, and thus, there is no deadlock situation possible in normal operation conditions. If one process dies (i.e. is killed, the machine hangs, etc.) the whole system falls in a deadlock condition. We will not consider here the reliability of the system. After all the necessary information is gathered, the processing can be performed, and the loop restarted.

The following pseudo-code represents the structure and logic of the *mat* program

```
process
      if parent()
            spawn partes*partes child processes
            assign every child process a piece of work
            for each row of children
                  do
                        for each children of the row
                              receive row
                              save to disk
                        end for
                  while not end
            end for
      else // this is a child process

            receive piece of work
            repeat cant times
                  send boundaries
                  receive neighbor information
                  doprocess
            for each row
                  send row to parent
            end for
      end
```

The parent automates the creation of children, distributing the work and collecting results. Children process is structurally simple: receive a piece of work and do a certain processing to it *cant* times. After each process finishes its looping, it starts sending the results to the parent. As soon as each finishes the communication, they quit. The parent collects the pieces of problem solved by each process, assembles them back in the right order and write the result matrix to disk. When this is done, the father process finishes its execution.

Before going on with the modeling of the system, there are some aspects regarding this particular scenario that are worth considering. We can see that every process has information needed by its neighbors and viceversa, it needs information from their neighbors. The amount of information needed is proportional to the submatrix size[24]. Each submatrix needs $\frac{2}{partes}(m+n+2partes)$ elements from its neighbors for each computation. The whole matrix needs $2.partes(m+n+2partes)$ individual element communications from neighbors so as to complete a time step and $2.cant.partes(m+n+2partes)$ for the total execution. On the other hand, the computation needed to solve a time step in a submatrix compromises $k\times\frac{m\times n}{partes^2}$ operations, where *k* is a constant determined by the *f* function. We can see that for a given $m\times n$ matrix, the amount of elements that have to be transmitted grows proportional to the square of the number of pieces *partes* that the matrix is split into. On the other hand, the number of operations needed to be performed by each processor decreases proportionally to the square of the number of pieces (*partes*) that the matrix is split into.

The model

We can see clearly that this problem falls within the class of SPMD parallel programs as the processing is the same, results are shared between processes and the role played by the parent process is the administrative role of process creation and solution assembly. As we saw on the taxonomy analysis for this kind of problems, there is no general network that can help us determining performance indexes. We need to first determine the parallel system in which the program will run and then we can obtain a Petri network where to simulate the real execution. First we need to determine a function *f* so as to have an algorithm to code, and thus, to instantiate the `doprocess` function so as to have a running program.
Lets call $n(x(i,j)t_0)$ to the set of adjacent elements in the matrix to a given element at time t_0. We will use the following function:

$$x(i,j)_{t_{n+1}}=\frac{1}{2}(x(i,j)_{t_n}+\frac{\sum_{y\in n(x(i,j)_{t_n})} y}{8})$$

This function performs a very local smoothing effect on each element of the matrix considering only the adjacent elements: it weights the average of all elements and the value itself of the point considered equivalently.
We will run this system using an image as the input matrix. The format of the image is 24 bits RGB (256 shades of red, 256 shades of green and 256 shades of blue per pixel), thus, the weighted average should be calculated for each color layer of each pixel. If we average the 24 bits number without considering the color layering, we get an undesired distortion of the image. The average should be calculated three times per pixel: once for each color layer. Another point to keep in mind is that the operations will involve integer arithmetic, since that is

24 Rightmost and bottommost submatrix dimensions must add the reminder of the integer division of ***n*** and ***m*** by ***partes*** to their height and width respectively. When ***partes*** does not divide exactly ***m*** or ***n***, the number of elements needed increase something.

the format of the image chosen.

We can see the chosen image of Garfield (© Jim Davies) as the input matrix. The image is 807 pixels in width times 976 pixels in height. The image format is 24 bits colour RGB[25]. The image was digitalized in black and white and later converted to the RGB format. All the pixels are either 0x000000 or 0xffffff. Each calculation has to be performed to each one of the $7.87x10^5$ pixels and has to consider all eight neighbors. We can see that there will be $7.87x10^5$ memory writes, $7.09x10^6$ memory reads and, at least, 11 arithmetic operations per pixel. These operations should not be understood as assembler or processor operations but high level ones.

There exists two numeric format conversions, one when the pixel is retrieved and another when stored into memory as a double precision float. In this case of RGB image these operations have to be performed on each color layer, thus, we have to separate the three layers, perform operations and then combine them back. We can then see that there have to be at least 40 elementary operations on each pixel, and thus, $3.2x10^7$ operations at least so as to compute every time-step. This lower level operations should almost match assembler or processor operations. We should also note that the matrix uses 6.0 MB of system memory when loaded as double precision float. No MMX extensions were used or considered. The goal here is not to obtain the best implementation of this problem for the specified system, but to have a tool for analysis.

Now that we have a fully defined algorithm, we shall model it according to our recommendation for SPMD class of problems. As there is no network that can model the general case, we have to explicitly define hardware configuration of the system that will run the problem.

We shall start addressing this problem with two CPUs: this is the simplest parallel scenario (more than one CPU) that we can consider. As our problem splits the original matrix in $partes^2$ pieces, we will choose the smallest *partes* that will partition the matrix in a number of pieces that is multiple of two: in this way we will allocate equal number of processes to each CPU[26]. As we saw before, there is very little overhead due to the allocation of multiple tasks to a single CPU, so sharing a CPU does not slow down the execution considerably. There is another important fact to consider, which is the fairness of the allocation of the CPU. We found that concurrent tasks with the same execution profile and same priority share the system resources fairly under Linux.

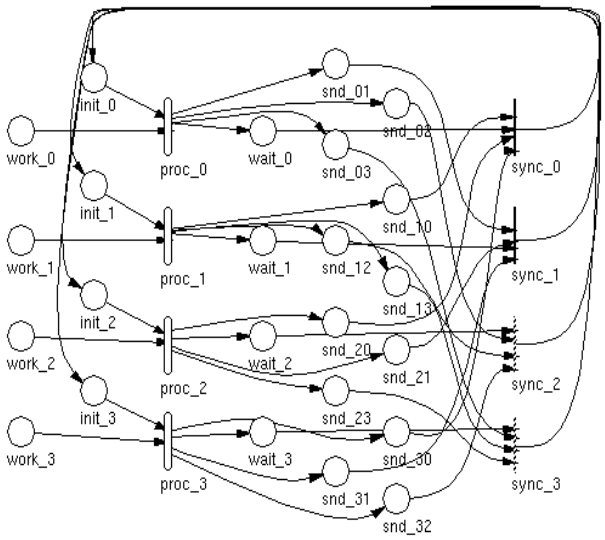
Based on the previous considerations, we will think of four CPUs, paired, each of them with half the power of the original CPU. We will obviate the overhead of context switches in this case. With this configuration, whichever matrix we address will be divided in four parts. Considering the wrapping of the general problem, we get the following scheme of neighborliness, and thus, process intercommunication:

25 We humbly accept the fact that the original Garfield paperback magazines came in black and white.

26 We will study later the allocation of single tasks to CPUs.

3	2	3	2
1	0	1	0
3	2	3	2
1	0	1	0

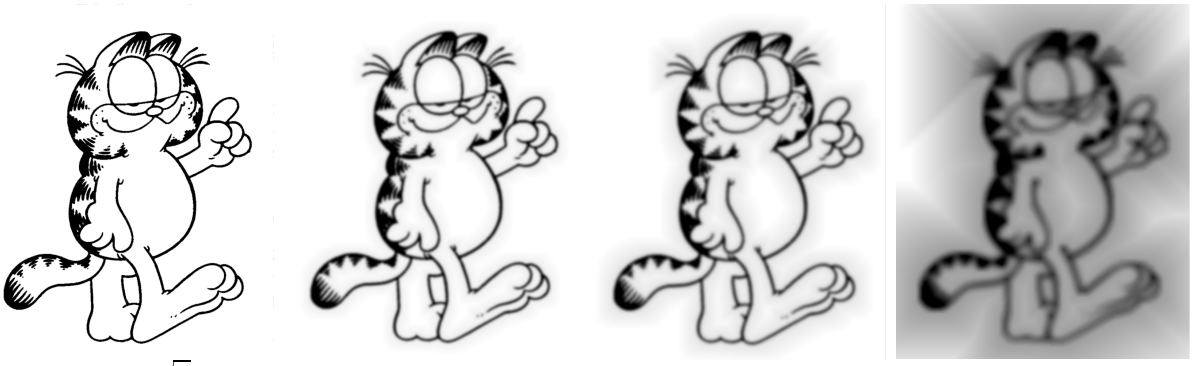
The white cells represent the matrix divided into four parts, while the shaded ones represent the neighbors due to the wrap. In this "extended" matrix we can see that all processes need to send information (twice) to all the others and need to receive information (twice also) from all the others. Let $\hat{P}$ be the set of submatrixes: $\hat{P}=\{0,1,2,3\}$. We define then the function $V::\hat{P}\rightarrow\widehat{P^n}$ based on the communication pattern in the way that it associates each element to the subset to its complementary subset on $\hat{P}$: $V=\{(0,\{1,2,3\}),(1,\{0,2,3\}),(2,\{0,1,3\}),(3,\{0,1,2\})\}$
Now we can apply the procedure described in the taxonomy section so as to define the associated Petri net to the algorithm. The following figure represents the resulting network:



We can see that there is much interdependence and interlocking at the communication stage, as we saw before. We now need to determine the computing power of each CPU so as to determine the remaining information and to fully define the Petri Net: we need to determine the distribution functions for each *proc* activity.

We will run the system on individual CPUs so as to determine their performance indexes. We will make successive terminating simulations, measure them and then, determine an index that represents the amount of work that the system can perform in a convenient period of time.
Before going on with the system analysis, we present here the result of the execution of *mat* on our example matrix.

What we have got here is the distortion produced on the image in 1, 30, 50 and 200 iterations respectively. We can see that this algorithm performs a blur operation on the image. We can see also a 45º darkening effect due to the equal propagation of the color to diagonal adjacent pixels. It is possible to smooth this effect with a more realistic effect weighting the diagonal



values with $\sqrt{2}$.

Parameter fitting

Experimental data

In this section we will determine empirically the needed parameters to complete the definition of the Petri Net. The way in which we determine the performance parameters is based on some particular features of both the application we are running and the input data: the parallel application can be run as a standalone program on a single machine without changes; the matrix we are using fits in system´s memory. These two facts allow us work on the same code and data that will run in parallel when measuring performance. This gives us a good level of accuracy.

We ran the simulation on x86 machines. The first system that we chose has a Pentium MMX processor running at 166MHz, configured with 96 MB of RAM memory. We have enough memory to hold the whole matrix in memory for the calculations since it takes 6 MB of RAM. The memory usage was obtained inspecting the OS's performance indexes. The code and data representation was not optimized for either this kind of application or data. We have to determine how fast this system completes the execution of a time-step and then, scale the problem so as to determine the average processing speed of the *proc* activities.

We ran our *mat* program twelve times with different iteration arguments so we can calculate individual iteration time. The magic number twelve proved adequate when we plot the results and observed very stable and smooth results. It is straightforward to see that the time behavior is linear on the number of iterations, so we model the curve of this experiment as a straight line $y=\boldsymbol{a}x+\boldsymbol{b}$ where x shall be the number of iterations and y is the elapsed time. In this coordinate system the x axis represents the amount of work while the y axis represents the time needed to solve it. In this scenario, we can see that the constant $\boldsymbol{a}$ represents the inverse of the processing speed (measured in iterations per time unit) while the constant $\boldsymbol{b}$ represents the time spent splitting and joining in the process. We can understand $\boldsymbol{b}$ as a fixed amount of time the system will be performing tasks before and after the process execution. We can see this tasks as work that has to be done (load into memory the parent process, process arguments, spawn children, communicate parameters, read matrix into memory, make the parent collect results from children, write results to disk) when no iteration takes place.

We took care to run the system on controlled conditions so as to get measures as stable as possible, to prove correctness of the model and also to be able to reproduce the execution conditions in further experiments. The system in which the process was run was not

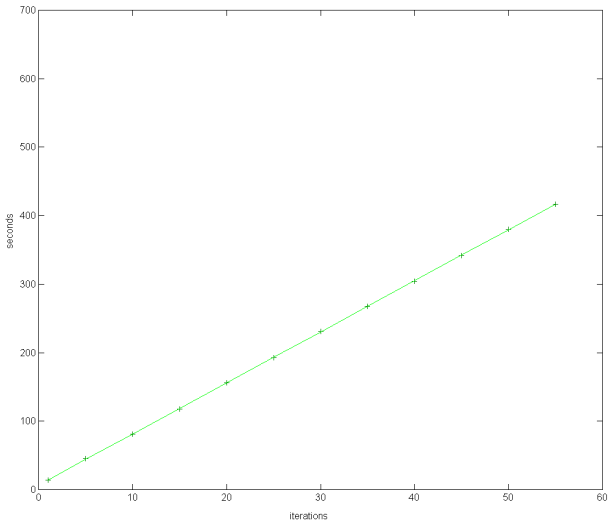
performing other task that operating system processes, that were also sleeping due to inactivity. No X server was running and the submission of tasks was done trough a Telnet session. No other users were logged or running processes at that time. No activity started by the cron ran at that time.
The following table shows the experimental data:

Iterations	time(mm:ss)	Iterations	time(mm:ss)
1	00:31	5	01:16
10	02:16	15	03:11
20	04:13	25	05:03
30	06:10	35	06:57
40	07:55	45	08:51
50	09:49	55	10:45

The time represents wall-clock elapsed time and not CPU allocation time. Using the least-squares method we determined that the curve that best adjusts to the data set is:

$$y = \mathbf{11.4}\, x + \mathbf{21.5}$$

where x counts iterations and the time (y) is measured in seconds. We can see that the processing speed of this system is around 5.28 iterations per minute (8.81×10^{-2} iterations per second) and that it takes almost 4.33×10^{-1} minutes (21.5 seconds) to split the process among the children and to join the answers back and write them to disk. According to our estimations, this 5.28 iterations per minute comprises 1.69×10^{8} operations per minute, 2.81×10^{6} operations per second.
The following graph plots the experimental data and the curve we determine. The plus signs represent the experimental data.



The second system that we chose is a Pentium Celeron processor running at 333MHz, configured with 128 MB of RAM memory. We have more than enough memory to hold the whole matrix in memory for the calculations since it takes 6 MB of RAM to hold the whole matrix. We also ran the system on controlled conditions so as to get measures as stable as

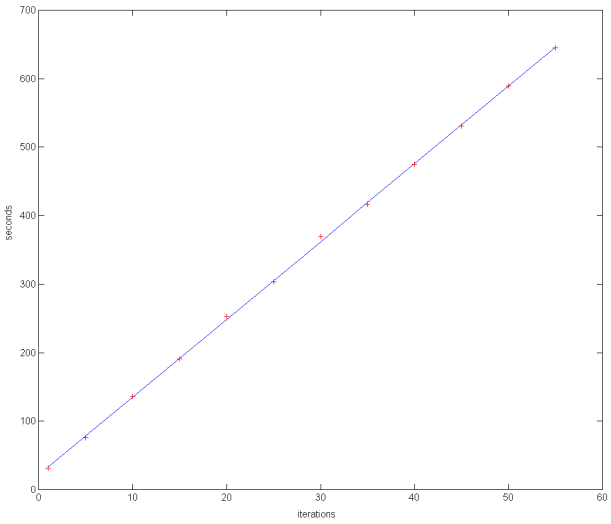
possible to prove correctness of the model and also to be able to reproduce the execution conditions in further experiments. The system in which the process was run was not performing other task that operating system processes, that were also sleeping due to inactivity. No X server was running and the submission of tasks was done trough a Telnet session. No other users were logged or running processes at that time. No activity started by the cron ran at that time.
The following table shows the experimental data gathered:

Iterations	time(mm:ss)	Iterations	time(mm:ss)
1	00:14	5	00:45
10	01:21	15	01:58
20	02:36	25	03:13
30	03:51	35	04:28
40	05:05	45	05:42
50	06:20	55	06:57

Using the root-min-square method we determined that the curve that best adjusts to the data set is:

$$y = 7.46\,x + 6.78$$

where x counts iterations and the time (y) is measured in seconds. We can see that the processing speed of this system is around 8.05 iterations per minute and that it takes almost 0.11 minutes (7 seconds) to split the process among the children and to join the answers back and write them to disk. According to our estimations, this 8.05 iterations per minute comprises 2.58×10^8 operations per minute, 4.3×10^6 operations per second.
The following graph plots the experimental data and the curve we determine. The plus signs represent the experimental data.



We can also note that processing speed does not scale well on MHz. Other processor and system's architecture details have to be considered so as to predict variations in speed basing ourselves on the processor. If systems processing speed was directly connected to the processors MHz, then our processing speed would be around 10.6 iterations per minute. We

only got 76% of that performance increase on the system.

Parameter determination

Based on the information we collected from the experiments we will calculate performance indexes for the Petri Net. Even though we have measured empirically the processing speed of our full-size problem on our target systems, that is not the size of problem that they will address when working in parallel: the parallel system will split the problem in four pieces and PVM's scheduling algorithm allocates two of them to each processor. Each portion of the problem will be one fourth of the measured problem and each processor would be addressing simultaneously two of these problems. With our rough estimation of the number of operations, we can say that the number of operations needed to complete each sub-matrix is of about 8 million operations.
It can be seen directly that the number of operations needed to solve this sub-problem are one fourth of the original one, and thus, the same CPU takes one fourth of processing time. On the other hand, the sub-problem will not be the only process running on the system: it will have to share the resources with one of his “brothers”. They have the same execution profile, they are both CPU bound and they both fit in memory simultaneously. As we saw when analyzing *primos* program, the fairness of CPU allocation is very high in this situation and we can think as of two independent CPUs, each with half the processing speed.
We will merge these two observations into single performance indexes. If we change the original work unit (the whole matrix) by one that is one quarter, it is quite straightforward to see that the processing speed will be four times the measured one. On the other hand, two processes will be spawned on each CPU, and thus, each process will receive half the processing speed of the CPU. We can conclude that the processing speed of each processor should double the measured speed. The distribution functions for the timed activities on our model are calculated on this basis.

We have gathered all the necessary information so as to model the parallel system. Now we have to determine what is going to be modeled by the tokens and the values for the distribution functions. As we said before, the tokens represent work units that are relevant to the problem. In our case, the work unit chosen will be the resolution of one fourth of the original matrix, or the matrix allocated to each CPU. When we initialize the Petri Net we will place as many tokens in the `work_i` places as iterations the system is going to model. The total number of tokens in the `work_i` places will be four times the number of iterations performed to the original matrix. `init_i` places are marked with one token, as the systems have only one CPU. The rest of the places count zero marks.
`proc_i` activities model our four “virtual” processing units, our two real multiplexed processors. As the sub-problem allocated to each processor is one fourth of the measured one, the processing speed of each of the original processors should be four times faster if we measure iterations per second. On the other hand, as the processors are multiplexed and their power is distributed equally, to both processes, so each virtual processor gets half the processing speed of the original. The resulting processing speeds are 10.6 iterations per second for the Pentium 166MHz system and 16.1 iterations per second for the Celeron 333MHz system.

Performance estimations

We shall now use our network to predict the performance of the parallel system. The problem itself of finding the most appropriate distribution functions might even imply a deeper study at this point, but we will use two distribution functions for our study: exponential and deterministic. As we stated before, they represent a pessimistic and an optimistic approximations. If the gap between these estimations is acceptable, then we can presume that the system performance will lie in between. These two distribution functions have also another benefit: we only need to determine one value to define them. Let's begin with the pessimistic estimation.

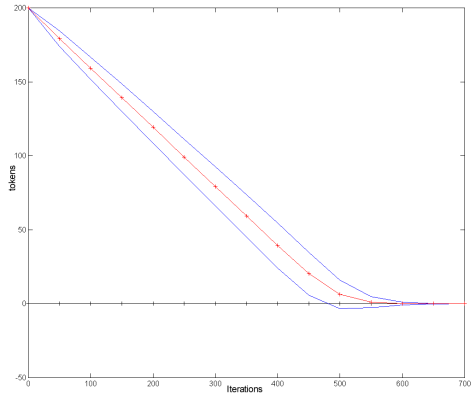
We have already determined the processing speed of our processors, or, what is the same, the time they spend solving each matrix. The figures are 11.4 seconds for the Pentium 166MHz and 7.46 seconds for the Celeron 333A processor. Our conceptual modeled system would consist of four processors, two with an estimation of 5.68 seconds per processed submatrix and another two of 3.73 seconds per processed submatrix. In this pessimistic scenario with exponential distribution functions, the rates would be $\frac{1}{5,68}$ and $\frac{1}{3,73}$ respectively.

We shall determine TET and MES for this system. It would be much easier if we had a tool that ran the system until it got to the steady-state and determines the elapsed time: the TET The tool we are using, the UltraSAN, is capable of determining the steady-state out of a Petri Net, but it does not account for the time, thus another method has to be applied. As we suggested before, we will use batch-means as a method for determining the TET

Lets assume that we plan to estimate the Total Execution Time of 50 time steps, then we have to place 50 tokens on each `work_i` place. We will execute successive terminating simulations until we determine that all the tokens were consumed. We need to monitor the number of remaining work units, for this reason we specify a performability variable that counts the number of tokens in the `work_ i` places. We will monitor the evolution of this variable through the different terminating simulations so as to determine when there is no more work to do, and thus, the processing has been done. From the single processor executions analyzed before, we can see that the Pentium 166MHz system solved 50 time steps in 9 minutes and 49 seconds while the Celeron 333A system did it in 6 minutes and 20 seconds, that means, 589 and 380 seconds respectively. We will study the remaining number of tokens with successive terminating simulations separated 50 seconds each. The following table summarizes the simulation data:

Iterations	Work remaining	Variance	Iterations	Work remaining	Variance
50	179,17	27,58	**100**	159,19	56,64
150	139,22	85,71	**200**	119,24	114,77
250	99,26	143,84	**300**	79,29	172,91
350	59,31	201,96	**400**	39,36	228,65
450	20,29	210,24	**500**	6,4	93,05
550	1,07	15,03	**600**	0.0897	1,03
650	0.0039	0.037	**700**	0.0001	0.0008

The following figure plots the data:



This plot was created with the following Matlab command: `plot(t,zeros(size(t)),'w-',t,zeros(size(t)),'w+',t,w+sqrt(v),'b-',t,w-sqrt(v),'b-',t,w,'r-',t,w,'r+')`, where the vector *t* represents the number of iterations simulated, *w* the expected values for the corresponding number of iterations and the vector *v* holds the values for the variance. The bounded connected component by the blue curves is the zone that holds the most likely values for the system performance.

From the graph it is easy to know that there exists a problem with the method: when we get close to the x axis, the linear behavior is lost. Apparently, the X axis becomes a limit for the expected value and, there is no expected zero value, but only arbitrarily small values. We used batch means trying to determine the time when we get to consume all tokens, but there is always a probability of having some token. This is because there is a very small probability of arbitrarily long execution times on every step due to the exponential distribution function. We can explain the source of error better with the following observation.

Given any arbitrarily big amount of time T, we can estimate a lower bound for the probability $P\left(\frac{T}{n}\right)>0$ that any time step can take longer than $\frac{T}{n}$ to complete its execution. Therefore, there is a non-zero probability that the whole execution takes more than any arbitrary big amount of time summing this arbitrarily long execution times of individual steps. This is not a consequence of the real phenomena we are studying, but a drawback of the distribution function we picked up.

We can now see that batch means alone is not a way to determine the TET for this problem. A workaround to this problem is to extrapolate the region where we observe the linear behavior and determine where it intersects with the X axis. We shall be able to say that the determined value is the TET for this problem.

From the inspection of the estimated curve, we can conclude that until t=450, the behavior can be considered pretty linear either for the red and the blue curves. Applying the root-mean-square method[27] to each of the curves we get the three following straight lines equations for the green and blue curves respectively:

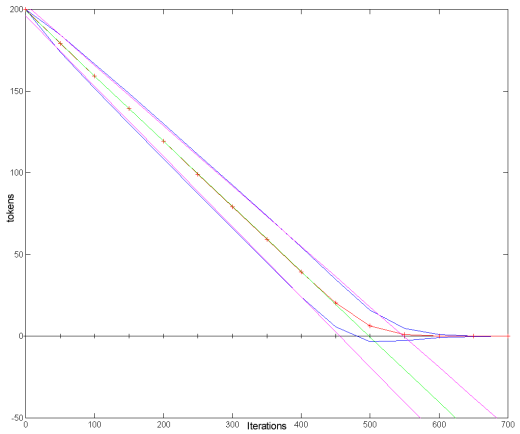
(i). y(x)=-0,399x+199
(ii). y(x)=-0,429x+196
(iii). y(x)=-0,369x+203

27 There is no specific background to model blue lines as straight lines but their shape and simplicity of linear approximation.

Equation (i) approximates the curve of expected values while equations (ii) and (iii) approximates the blue curves. The solutions, $\frac{x}{y(x)}=0$ for each of the equations follows:

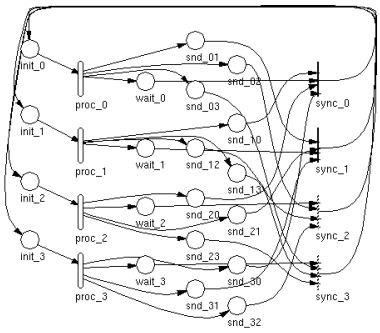
(i). x = 499
(ii). x = 456
(iii). x = 549

A joint plot for all the curves follows:

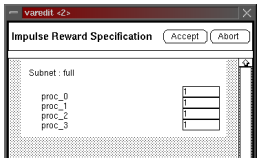


We can determine [456 ; 549] as the mostly probable interval where the execution time of the real experiment must fall, while 499 seconds or 8m18.99s is the expected Total Execution Time. From this plot we can also get another estimation: the mean execution speed. It is straightforward to realize that the tangent of the first line is a good estimation for the MES. Remembering that the tokens used in the simulation represent one quarter of the original matrix, we can say that the MES of this system is 0.399 submatrixes per second or, what is the same, 9.98×10^{-2} matrixes per second. A better time unit for expressing the processing speed is the minute. The processing speed or MES is of 5.99 matrixes per minute.

Another way to estimate the TET comes out of a different approach. We determined the TET first and after that we determined the MES. We will try now to determine the MES of the system and then the TET.
As we saw before, there was a systematic error in the previous procedure, an error that becomes relevant when we try to determine TET itself, but that is not a drawback for finding a stable stage that allows us to extrapolate the behavior. We want to determine the processing speed of our system during that stable stage. As we suggested in previous chapters, we can modify the Petri net that represents the problem by removing the `work_i` places. We can see that the resulting Petri net will loop forever without any fixed token configuration: all tokens will loop their own path. The following diagram represents the resulting Petri net.



We want to determine the MES for this Petri net. We need to account the number of times the



tokens cycle the network in a period of time so we can average the processing speed: the MES We defined a performability variable *vueltas* that is associated to the processing activities, signaling a unitary value with each submatrix processing. We shall then simulate the execution with the Accumulated Reward Solver simulator provided by the UltraSAN tool. The number obtained is the number of submatrixes solved within a certain period of time, four times the processing speed of the system in that period of time, as a full original matrix is represented by four of these tokens.
The result of the simulation after 10^4 seconds is $3.99x10^3$ tokens, $3.99x10^{-1}$ tokens per second, $9.99x10^{-2}$ matrixes per second or, expressed in a more convenient unit of time, 5.99 matrixes per minute. The similitude with the previous estimation is remarkable.
It is important to note that it is valid to associate 4 tokens, one in each `init_i` place to a matrix without losing the resolution semantics of the resolution due to the synchronization performed at `sync` activities. This synchronization ensures that no other portion of the problem is solved until the directly connected places finishes their work. We can see that all four processors will work tightly coordinated, waiting for the slowest one after the completion of the assigned piece of work: each process waits for the information form its adjacent before iterating again one step of time. This behavior is the same all the execution long, and, at the instant of time 10^4 some processes might be waiting for others, but all of them will be either solving the same time-step or they will be waiting for the others before computing the next. It is under these considerations that computing back from tokens to matrixes is valid.

We now need to determine an optimistic estimation for the system performance. As we stated before, we shall use deterministic distribution functions to estimate TET and MES for our system configured for running the previous matrix fifty time steps. The simulation tool we are using does only provide simulators for determining the steady state out of a network but not the elapsed time. Furthermore, if the net contains a distribution function that is not exponential or instantaneous, transient measures cannot be obtained. We shall now study the evolution of the network analytically.
We are assuming that there are fifty tokens into each `work_i` place and one token into each `init_i` place at time t_0[28]. Lets assume that `proc_1` activity and `proc_3` have deterministic values 5.68 while `proc_0` and `proc_2` have values 3.73. All `proc_i` activities are enabled and thus, their execution begins at time t_0. At time t_0 + 3.73 both activities `proc_0` and `proc_2` complete their execution and individual tokens are placed into `wait_0`, `snd_01`, `snd_02`, `snd_03`, `wait_2`, `snd_20`, `snd_21` and `snd_23` places. As no activity remains enabled the state

28 We assume that all remaining places have 0 marks.

remains the same until time $t_0 + 5.68$ where activities `proc_1` and `proc_3` become enabled. Tokens are then moved to places `wait_1`, `snd_10`, `snd_12`, `snd_13`, `wait_3`, `snd_30`, `snd_31` and `snd_32`. At this very moment, activities `sync_0`, `sync_1`, `sync_2` and `sync_3` are enabled. Tokens are removed from all previous places and individual tokens are placed back in `init_0`, `init_1`, `init_2` and `init_3` places. We can see that current system state differs from the initial state because it holds 49 marks into each `work_i` place instead of 50. We can also see that if we call t_i' to time $t_0 + 5.68$, we can see that at time $t_i' + 5.68 = t_0 + 2 \times 5.68$ there will be 48 marks into each `work_i` place.
We can see that tokens are consumed 4 every 5.68 seconds, or one every 1.42 seconds: 42.3 tokens every minute. As we saw before, each matrix consists of 4 tokens, thus, the predicted processing speed, the MES is of 10.6 matrixes per minute.
Now that we know the MES, we can estimate the time it will take the system to consume all tokens. Knowing that the system holds 200 tokens, representing 50 matrixes, it will take 284 seconds, or 4m43.90s, the TET
It is also quite noticeable why the slowest CPU is driving the performance of this kind of parallel systems. We can see that during the interval ($t_0 + 3.73$; $t_0 + 5.68$) the only CPU activity comes from the slowest CPUs while the faster ones remain idle, waiting for the slow to finish. It is straightforward to see that this computation pattern prevails even when the number of CPUs and work units grow. The execution of faster CPUs will be "bursty" and periodical, while slowest CPUs will run continuously. It is also noticeable that the effective processing speed of all CPUs shall be equal to the processing speed of the slowest, thus, the optimistic approach for this case suggests that the processing speed of the system shall be at most as fast as *n* times the speed of the slowest CPU, the weakest link, where *n* is the number of CPUs in the system.

Now we have an upper and lower bound for the expected system performance based on the pessimistic and optimistic estimations obtained before. We have determined the following intervals for our performance parameters:

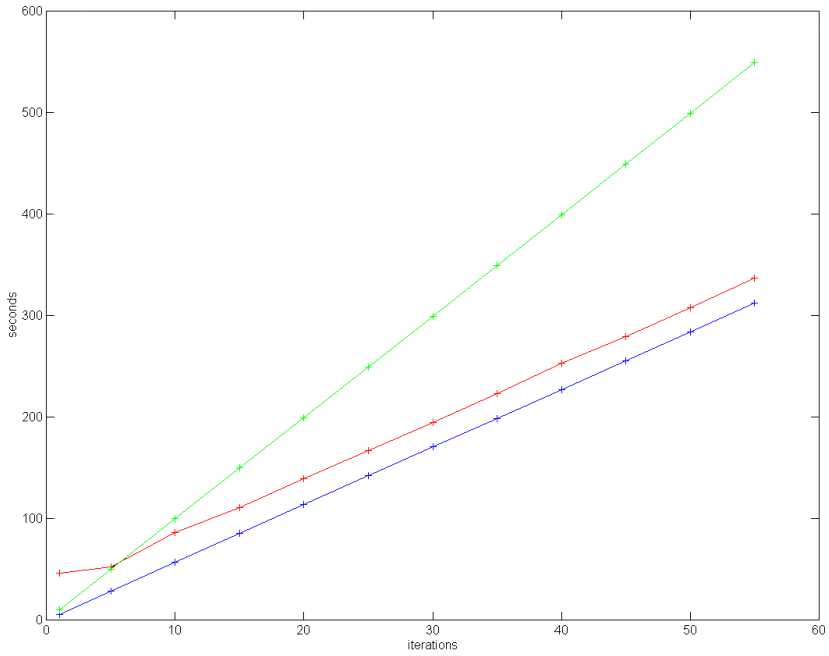
TET	284s – 499s
MES	5.99 – 10.6 matrixes/minute

We would like now to test the estimated performance against our measures from the real system execution. We run once again the `mat` program, but this time, with PVM's virtual machine configured for running over the two processors. The execution time was of 5 minutes and 8 seconds (308 seconds), which falls within the predicted interval. The MES for this system is of 1.62×10^{-1} matrixes per second or 9.74 matrixes per minute, value that also falls within the expected interval for the performance parameter. It is also important to note that the estimated performance falls within the first quarter of the predicted interval, closer to the optimistic estimation. It is a reasonable behavior as both systems were devoted to this task: no other process was run meanwhile.

We repeated the previous tests on our system, for a broader set of intervals ranging from one to 55 iterations, as for other measures acquired. The following table shows the collected data from the execution:

Iterations	time(mm:ss)	Iterations	time(mm:ss)
1	00:46	5	00:52
10	01:26	15	01:51
20	02:19	25	02:47
30	03:15	35	03:43
40	04:13	45	04:39
50	05:08	55	05:37

The following graph plots all the collected data against individual system performance and estimated performance boundaries given by optimistic and pessimistic estimations:



We can see the linear behavior found again in this observation, plotted in red. The blue and green lines plots optimistic and pessimistic estimations respectively. We shall mention that in this scenario where two systems are involved we can observe a much higher initialization time than in the single CPU scenario. There is an extra load on the file server at startup, when it has to serve the program and the data to both CPUs. It is also noticeable that the experimental behavior is reasonably similar to the one of the optimistic estimation; it is something that we expected from a dedicated system. The model represents the behavior of this system properly.

Limits for the model's predicting capabilities

We shall discuss some aspects to be considered when using the previous models regarding their predicting capabilities. We will try to be practical when considering this matter.
There are several considerations regarding the individual system performance, which are beyond the scope of this study, but should not be omitted in order to get accurate estimations. The concept “system performance” is difficult to ascertain, but we can simplify it just considering it as how fast a system can perform certain task for us. This simple assumption is

compatible with our system performance concept for the parallel system and is good enough for individual system performance.
It is very difficult to speak about system performance (generally wrongly associated to MHz) without considering any particular task. According to our very informal definition of performance, we cannot speak about performance without specifying a particular task. There exists also a problem when we try to compare performance but measuring different tasks[29]. Shouldn't we have a single performance indicator for a system regardless the task we are considering? We believe that it is not correct to go that way. Lets think two different implementations for a same problem, one that is 8086 compatible and another that uses MMX if available, extensions for video decompression. If we are comparing two systems *a* and *b*, *b* has a clock that is 16.66% faster than *a*, same vendor, but *a* with MMX extensions and *b* without them. If we compare them according to the first implementation it is most likely that *b* proves faster than *a*, but if we compare them according to the second implementation, the opposite result is the most probable. It is very unlikely that we can describe all the parameters regarding system performance with a single figure. If we gathered only one performance index for a system then we would not be able to make a difference if the problem is optimized for certain kind of operations or not: either *a* would be faster/slower/equivalent to *b* for all tasks, but we can see that it does not model all possible performance behaviors.
Another problem arises when we are considering both different systems and different tasks. This is the case we face when we use our models to predict the performance of the parallel system: the tasks will be different (at least the input problem would be scaled) and the first test systems themselves will be, most probably different[30] to the production ones.
We shall now introduce an informal concept: *input-execution equivalence*. The aim is to partition the task space according to its execution profile, based on their execution and IO blocking interleave. Interruption-level events, memory swapping, IO were discussed when analyzing distributed.net's approach to the RC5 challenge and were discarded out of the model and summarized into the distribution function that models the process activity. The fact that these factors were not modeled does not mean that they are not important. They are too small to be considered in out Petri network but they have to be taken into account when estimating performance: if possible, they should remain invariant in the benchmarked systems and the constructed parallel system.
Some metrics could be developed to quantify these concepts like number of IO requests per time unit, number of swap-in, swap-out pages per time unit, etc. but are beyond the scope of this analysis. To be able to exhaust the different levels of input-execution equivalence, locality has to be taken into account. Even though there might be no interrupt level blocking due to IO operations, there is another kind of blocking, this time caused by pipeline stalls due to memory access when cache misses occur. Whenever we cross a border of the memory hierarchy of a system there is a tremendous performance price to pay. The following table summarizes few performance data gathered from some of our test systems[31]:

29 We will consider different tasks same algorithms applied to different data sets.
30 Lets remember that this tools would most probably be used when designing the parallel system rather than first buying nodes and later benchmarking.
31 The measures were gathered using memtest 2.75 and are intended ***only*** as examples just to show some empirical data.

Processor		Celeron 333 MHz	PII 400 MHz	Celeron 800 MHz
System clock		66MHz	100MHz	100MHz
Cache L1	Speed	3300 MB/s	4100 MB/s	6300 MB/s
	Size	32 KB	32 KB	32 KB
Cache L2	Speed	830 MB/s	590 MB/s	950 MB/s
	Size	128 KB	512 KB	128 KB
DRAM Memory	Speed	61 MB/s	88 MB/s	158 MB/s

We can see that there is about one order of magnitude of memory bandwidth loss when we miss on L1 cache and another order of magnitude when we miss the L2 cache. This means that if our loop fits inside the L1 cache, it can produce/consume data at a rate of gigabytes per second, if it fits the L2 cache, it can produce/consume data at a rate of hundreds of megabytes per second while if our data set is so big that we always get cache misses, our data produce/consume rate would be of tens of megabytes per second. The case of virtual memory and disk access to retrieve a virtual memory value is even worse. Even though today hard disks can transfer data at a speed equivalent to the memory, the seek latency (which is measured in *milliseconds*) and the block transfer throws down the effective speed to thousands, tens or even few kilobytes per second.

On the other hand, knowing more details about our code can help us decide which processor is "better" for our application. For example, lets assume that our code is 90% of the time executing a loop that is 350KB long. Lets also assume that there is no inner sub-loop that presents a local behavior. For both Celeron processors in our figure, data access would mean memory access, and thus, the speed would be around 100 MB/s. On the other hand, that piece of code would fit in the Pentium II L2 cache, which would yield at least four times the memory bandwidth of the DRAM memory. For this particular case, it seems that the 400 MHz processor would perform faster than the 800 MHz processor. Once again, these issues are highly coupled to the problem itself and cannot be separated from it.

With the concept of *input-execution equivalence* we try to consider these facts as much as possible out of different program execution and try to state that both executed in pretty equivalent conditions. What would be the point of estimating the performance of a parallel system if we measured individual systems running at L1 speed while the estimated cluster will do excessive swapping?

Lets see the impact of these considerations with some figures. We will try to force our test systems across some of this performance boundaries. We produced a set of input matrixes with the following dimensions: 500x500, 1000x1000, 2000x2000, 3000x3000 and 4000x4000, which were run 5 iterations on our mat program. The following table summarizes memory usage per CPU and elapsed execution time:

	Pentium 166MHz	Celeron 333A	P 166 + Cel 333A
	96 MB RAM	128 MB Ram	96 + 128 MB RAM
500	47 s	24 s	26 s
memory (MB)	3,83	3,83	1,91 + 1,91
1000	187 s	96 s	108 s
memory (MB)	15,29	15,29	7,64 + 7,64
2000	738 s	387 s	403 s
memory (MB)	61,1	61,1	30,55 + 30,55
3000	4424 s	2028 s	828 s
memory (MB)	137,42	137,42	68,71 + 68,71
4000	-	-	3723 s
memory (MB)	244,26	244,26	121,13 + 121,13

The basic calculations on the memory used by the process data considers two complete matrixes of double precision real numbers. There is no elapsed time measure for the individual systems with matrix of 4000x4000 elements because the runs did not finish: Pentium's hard drive failed after ten days of processing and we did not let the process run on the Celeron for

more that 5 days. In either case, hard drive led was all time red due to permanent swapping (trashing describes better the situation) and CPU allocation to the task was at most, 1 or 2 percent of the time. We can see that while the problem fitted in RAM memory, the elapsed time grew quadratically on the size of the matrix (for individual systems, 500, 1000 and 2000 elements). For 3000x3000, as swapping starts, there is an over-quadratic growth on the execution time. This is due to a change in the *input-execution* profile: when data is needed and page faults occurs, execution is blocked until some page is removed from the RAM, written to disk, the page is tagged as available and the needed data is retrieved from the hard disk. Due to the high frequency of these faults, the performance drops.
It is also good to notice that on the combined system, the elapsed time still grows quadratically for 3000x3000 matrix: sub-problem still fits in RAM.
Lets assume that we are trying to predict the combined system performance for the execution of a 3000x3000 elements matrix. Lets assume also that we did not considered the concept of *input-execution equivalence* and that we made individual system performance measures with 3000x3000 input matrixes. After simulating the Petri Net, we would have predicted an optimistic execution time of 2190 seconds, while the observed execution is of 828 seconds, almost three times shorter. This is due to the lack of *input-execution equivalence*. If we predict using the 2000x2000 single processor execution times, we get the right estimation. We can see that memory usage of the 2000x2000 matrixes in the individual systems is similar to the memory usage of each individual system on the 3000x3000 experiment on the combined system.
The lack of quadratic growth in the elapsed time for the 4000x4000 experiment is due to the growth in the memory usage, specially on the Pentium system: swapping was needed.

There exists another important reason for execution stalls that arises due to excessive communication amongst processing nodes: if the ratio between remote and local data needed to perform a processing stage is not reduced, the amount of time spent in communication threatens the time won parallelizing the application. Whenever we have to access to information that is stored on other process memory, the data transfer speed drops compared to memory access speed. Even if we are using a high speed network (i.e. 10 Gigabit Ethernet) what we can do is to reduce transmission time, but there is an important price to pay in O.S. calls, switch contexts memory access blocking, etc. It is difficult to minimize latency. We have to balance the amount communication time vs. processing time.
We ran again simulations with mat program, but this time fixing the size of the problem and varying the number of processes solving the problem both on individual systems and on two computer system. We solved a 500 x 500 matrix over 20 time steps using 1, 4, 16, 36, 64, 100, 144, 196 and 256 processes. Unfortunately we had no massive parallel system to run our tests, so most of the communication happened over UNIX sockets instead of TCP/IP ones, but we were able to see the effect of communication on the processing time anyway.
First of all we see the amount of communication that takes place for each experiment. The following equation approximates the number of cells exchanged on every iteration, where i is the number of parts (horizontal and vertical) in which the input matrix is going to be diced:

$$4*(\frac{500}{i}+1)*i^2$$

We gathered data from our test systems, the Pentium 166 MMX and the Celeron 333A alone and then solving together the problem using both a 10 Mbps and 100Mbps Ethernet LAN.

When solving the problem using both systems, experiments contained more than one process. The following table shows the experimental data:

Partes	Pentium 166MMX			Celeron 333A			Lan 10			Lan 100		
	Elapsed time (mm:ss)	Per process communication overhead (s)	Estimated execution time (s)	Elapsed time (mm:ss)	Per process communication overhead (s)	Estimated execution time (s)	Elapsed time (mm:ss)	Per process communication overhead (s)	Estimated execution time (s)	Elapsed time (mm:ss)	Per process communication overhead (s)	Estimated execution time (s)
1	01:37	0	97	00:58	0	58						
2	01:43	1,5	24,25	01:01	0,75	14,5	00:51	0	12,75	00:51	0	12,75
4	02:17	2,5	6,06	01:13	0,94	3,63	01:07	1	3,19	01:08	1,06	3,19
6	03:11	2,61	2,69	01:30	0,89	1,61	01:37	1,28	1,42	01:37	1,28	1,42
8	04:30	2,7	1,52	01:56	0,91	0,91	02:29	1,53	0,8	02:30	1,55	0,8
10	06:02	2,65	0,97	02:30	0,92	0,58	03:23	1,52	0,51	03:23	1,52	0,51
12	08:12	2,74	0,67	03:10	0,92	0,4	04:28	1,51	0,35	04:27	1,5	0,35
14	10:24	2,69	0,49	03:58	0,92	0,3	05:42	1,48	0,26	05:40	1,47	0,26
16	13:29	2,78	0,38	04:55	0,93	0,23	07:15	1,5	0,2	07:13	1,49	0,2

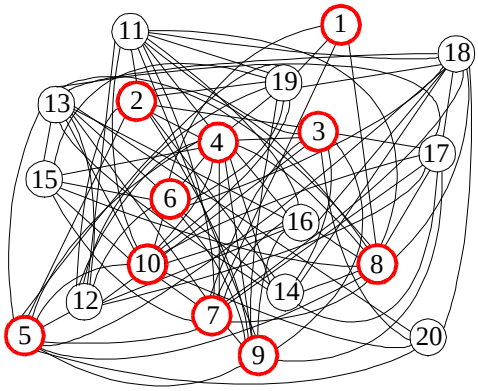
It is easy to note that in all cases, as the number of processes grows, the elapsed time grows too. We added a column, "Per process communication overhead" that calculates the overhead introduced by each process, considering that a single process has 0 overhead and understanding process overhead as $\frac{singleProcessTime - experimentTime}{numberOfProcesses}$. We can see that as the number of processes grows, the per-process overhead somehow stabilizes[32]. We can see that in the multiple systems experiment the network speed has almost no influence on the execution time. In this case, the overhead is almost completely due to OS calls and protocol overhead. We used the same NICs so as to keep the driver overhead time constant.

These figures help us understand that we have to pay a price in terms of overhead whenever we add a process, and it has to be worth doing so. We can see that it is easy to spend more time communicating than processing, in other words, for each problem, system and algorithm, there is a speedup limit that can be obtained out of parallelism. Also, we can figure out that adding CPUs might not always result in speedup: if the overhead introduced is bigger than the speedup, then no speed is gained.

32 This behaviour should be somehow consistent while the input-execution equivalence prevails.

5.3 - Task-Farming example application: *SN metaheuristic*

This JAVA application implements the ***SN*** metaheuristic proposed by Sebastian Urrutia and Irene Loiseau [URR1] to solve the *Steiner Problem in Graphs* (**SPG**).
The **SPG** consists of finding a sub-network that covers a subset of nodes of a given network with minimum cost. **SPG** models adequately communication scenarios, specially multicast ones. Research on the field was pushed in the last decade by the telecommunication industry. Sometimes the **SPG** problem is also referred as SPN or Steiner Problem in Networks.
A formal definition of the problem follows:
Let *G=(V,E)* be a connected undirected graph, where *V* is the set of nodes and *E* denotes the set of edges. Let *w* be a non-negative weight function $w:E \rightarrow \mathbb{R}+$ that associates the set of edges with positive real values and a let X be a subset $X \subseteq V$ of nodes called ***terminal nodes***. Let $n_X = |X|$ be the number of terminal nodes. The Steiner problem **SPG**(*V*, *E*, *w*, *X*) consists of finding a minimum weight connected subgraph of *G* spanning all terminal nodes in *X*. The solution of **SPG**(*V*, *E*, *w*, *X*) is a Steiner minimal tree (SMT). The non-terminal nodes that are part of the solution are called *Steiner nodes*. This problem is inherently complex (from the point of view of computation). Karp proved that **SPG** is NP-Complete in the general case, thus, a fast parallel metaheuristic solution is helpful in the field.
Lets introduce an example of a small SPG. The graph used, a HEIDI graph named `gD-T1a10.exp`, which has 20 nodes. The following is as graphical representation of it where circled in red nodes are terminal nodes.

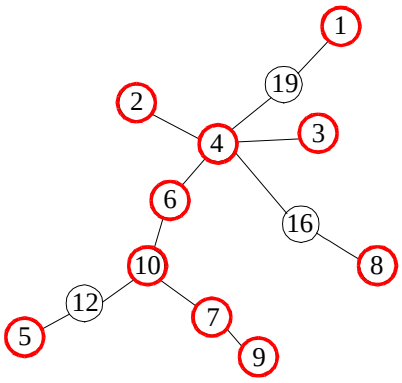


The following table defines *w* function for all edges:

edge	w	edge	w	edge	w	edge	w	edge	w	edge	w
17-8	0.798440	19-12	0.858676	18-20	0.497259	11-7	0.909643	19-4	0.225491	16-12	0.477361
5-7	0.277775	19-9	0.684219	17-18	0.649707	17-10	0.180421	7-8	0.344251	19-15	0.438562
10-13	0.513401	10-5	0.920128	13-5	0.316867	15-4	0.492422	14-2	0.778257	18-16	0.536742
18-13	0.141603	16-7	0.226107	5-2	0.223656	4-8	0.931895	9-7	0.230996	10-19	0.884318
1-6	0.804177	6-12	0.169607	11-19	0.546107	13-17	0.496074	11-4	0.014579	18-5	0.865434
9-3	0.998925	10-15	0.935004	3-10	0.944318	8-14	0.258906	11-18	0.809785	12-2	0.563617
11-17	0.296032	8-15	0.294160	7-17	0.003231	2-3	0.363598	7-13	0.532960	1-8	0.923692
11-10	0.292517	16-4	0.074530	12-17	0.675476	7-3	0.934495	11-9	0.718867	10-6	0.122326
11-16	0.891529	15-13	0.639458	10-7	0.182556	9-17	0.244327	1-7	0.655368	4-5	0.176239
8-16	0.069755	3-4	0.078232	2-9	0.673936	4-9	0.783282	4-7	0.879009	6-18	0.190709
11-3	0.663227	2-4	0.204329	8-5	0.627158	10-20	0.210883	7-18	0.157139	9-16	0.356383
8-2	0.457702	3-20	0.870540	13-6	0.087644	17-3	0.382896	17-14	0.191112	6-4	0.058052
6-19	0.850920	1-19	0.180372	18-14	0.111276	13-16	0.051508	5-19	0.933420	19-2	0.469050
11-8	0.512535	8-18	0.359095	14-6	0.288379	10-16	0.442560	13-20	0.324541	12-4	0.476355
11-2	0.931835	12-10	0.099640	5-20	0.827391	11-12	0.304285	12-13	0.775421	14-13	0.269022
15-6	0.639979	15-7	0.576971	5-9	0.918930	10-4	0.525747	8-13	0.143982		
14-4	0.880075	19-18	0.979434	14-3	0.383832	12-5	0.070090	6-9	0.717597		

Given all the previous data we determined that the optimal solution cost is 1.5963689, and that

nodes 19, 16 and 12 are Steiner nodes. The following graph is the solution:



The ***SN*** metaheuristic transforms the **SPG** into an iterating set of decision problems based on **SPG** heuristics. Each of these iterations is highly decoupled and can be easily parallelized, as the authors suggest on [URR1].
The idea behind the *SN* metaheuristic is both simple and powerful as it turns optimization problems into decision ones extracting information for successive decisions from possibly inaccurate results of heuristic resolution. The following pseudo-code is extracted from [URR1] and describes SN's basic scheme:

While it is possible to divide Q into subproblems $q_1..q_n$ do
For i from 1 to n do
Solve q_i heuristically to obtain s_i and cv_i
End For
Obtain i with maximum cv_i
Modify Q using the information provided by q_i and s_i
End While

The method assumes that the problem **Q** can be divided into **n** subproblems $\mathbf{q_1,...q_n}$. Using the heuristic we can obtain a certain solution $\mathbf{s_i}$ with an associated cost $\mathbf{cv_i}$. After determining the optimal heuristic solution, the problem **Q** is modified into **Q'** that should be simpler, according to some heuristic's metric.
The *Departamento de Investigación Operativa* at the *Instituto de Computación, Facultad de Ingeniería* has developed a C++ tool for graph handling: Heidi. The development of this line of investigation is founded by CONICYT. The tool has been evolving since 1993 through successive individual grade students thesis works. This evolution started with a graphical system based on Motif and Sun's C++ compiler. Each successive work added not only functionality, but new implementations for the graphs and translations from one stage to the other.
One approach to our development could have been choosing the most adequate graph classes in C++ and develop it within Heidi's environment, but that would have constrained us to Sun systems. This is fine within Heidi's framework, but this research has to consider specifically COTS hardware and software. Even though Solaris runs on PCs, it is not either a standard parallel or a commercial environment. We could have developed a set of PVM or MPI routines from scratch and interface them with C++ graphs from Heidi. In this approach we could obtain portability, even through different platforms, but the marshalling and unmarshalling of information would be our responsibility: we would have to consider aspects like big & little

endians and bitwise operations, data formats, etc. Message passing libraries offer a very low level support for information interchange, not adequate for complex data structures like graphs. We followed this approach with the *Mat* application, but the information interchanged were vectors of double precision float numbers.
Most of these problems are solved using Java, a younger alternative to Object Oriented Programming. Java is inherently platform independent and the bytecode can be run in any system where exists a Java Virtual Machine. A significant drawback is that the JVM's operations, the bytecode, is interpreted, and thus, slower than the object code produced out of a C++ compilation. Other compilers produce executables with the JVM embedded that can be run directly as a regular application, but loosing portability. At this point we decided to work with Java. The interface with Heidi is the `edu.fing.inco.math.util.HeidiGraphFileIO` class that reads and writes files with graphs understandable by Heidi.
The problem of exchanging information between programs is redefined in Java as referencing remote objects using RMI (Remote Method Invocation). A simple way to think about this problem is that one object happens to live in another machine, and that you can send a message to that object and get a result as if the object lived on your local machine. RMI makes heavy use of interfaces. When a remote object is created, the underlying implementation is masked by passing around an interface. Thus, when the client gets a handle to a remote object, what it really gets is an interface handle, which *happens* to connect to some local stub code which talks across the network. The only difference is that the remote object is *bound* to a variable instead of *created* as a regular object. From then on, it behaves as any regular object. Another important factor is that most objects can be used this way. The only requirement is that the object implements the `java.lang.Serializable` interface and uses Serializable objects.
Since Java 1.1 object serialization was introduced. It makes it possible to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be fully restored into the original object. This is even true across a network and different Java Virtual Machines, which means that the serialization mechanism automatically compensates for differences in operating systems and hardware platforms. Serializing an object is quite simple, as long as the object implements the **Serializable** interface (this interface is just a flag, and has no methods). In Java 1.1, many standard library classes have been changed so they're serializable, including all the wrappers for the primitive types, all the collection classes, and many others.
A particularly clever aspect of object serialization is that it not only saves an image of the object directly referenced, but it also follows all the handles contained in the object and saves *those* objects, and follows all the handles in each of those objects, etc. This is sometimes referred to as the "web of objects" that a single object may be connected to, and it includes arrays of handles to objects as well as member objects. More details can be found on the Java documentation (http://java.sun.com).
We can see the importance of this approach: as far as we can serialize all our objects, we can run them remotely on any remote Java Virtual Machine. We can concentrate on graph algorithmic instead of communication and synchronization.
Another reason is the possibility of having a multi-platform parallel tool available for running experiments and test our models also in a non-standard parallel environment. Our analysis considers the platform, the algorithm, the communication pattern, etc., but does not constrain to standard parallel environments.

The SN algorithm is implemented as a method in a class. It receives a weighted graph, the subset of terminal nodes, a heuristic, a criteria and returns the resulting tree. The metaheuristic

converts the problem into a succession of sets decision problems: at each iteration, the “best” decision is taken, until the solution is found. This general metaheuristic can be applied to almost any problem that accepts a compositional solution. In the case of the **SPG**, the decision consists of determining at each stage for every non terminal node will or will not be part of the solution. Each individual decision is taken considering the heuristic applied to the graph considering that the decision has already been taken. The “best” option, according to the heuristic, is taken at each stage, “fixing” each non-terminal node as a Steiner node or removing it from the solution. Let $m=|V \setminus X|$ be the number of non-terminal nodes in the graph. Each iteration invokes twice the heuristic for each non-terminal node at each stage. We can see that the full solution of the problem takes $m(m-1)$ invocations of the heuristic.

The following pseudo-code represents the implementation:

```
process
      initialize remote object threads
      for (i=0;i<cantTermNodes;i++)
            build set of graphs
            apply remote object threads to set of graphs
            pick best solution
            replace current graph with best graph
      end
      determine min coverage tree
```

Java makes the use of multithreading simple. We use a thread to control each remote object. The threads access a common set of graphs to solve, pick some of them, submit the job to the remote object, gather the result and send more jobs until no other graphs are there for solving using the heuristic. The resolution of the heuristic takes place remotely, but the synchronization and access to the information is solved within the same Virtual Machine, which makes it simpler to coordinate execution. Instead of having different programs running in different memory spaces, we have a set of threads running with the same permissions in the same virtual machine. The set of remote objects do not interact amongst them, but through the master process.

Complexity

The way we parallelized the algorithm, the same suggested by the article's authors, consists in running in parallel all the heuristics that cooperate to take each decision. The most simple approach is to take single CPU heuristics and run them in parallel. This is the approach we followed. We used single-threaded heuristics to solve each decision problem, while running sets of them in parallel[33]. The metaheuristic imposes a limit in the speedup: we can not spawn more than 2*m*, twice the number of non-terminal nodes, decision problems at the same time, even though there are *m*(*m*-1) heuristics to solve. We have to take one decision at a time so as to build the solution. At the following iteration there will be two heuristics less to run, we already took a decision. Each iteration will require less computations to solve, until the last

33 The validity of this analysis remains even if the heuristics are solved in parallel by fixed sets of computers also.

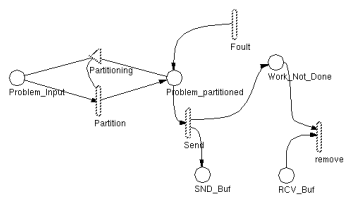
decision when we have to decide if the remaining node shall be present or not in the solution, leading us to computing two heuristics. We can see that the usage of computational resources decreases in time. If we have as many as 2*m* CPUs, only one iteration would use them all. Following iterations would use successively two CPUs less than the previous iteration.
It is evident now that even if we count with *m*(*m*-1) CPUs we will not be able to solve all heuristics at once. A first lower bound for the speedup of the problem is given by the sequence of stages. If a single system attempts to solve the problem, it will have to solve *m*(*m*-1) heuristics. If we have enough CPUs to tackle all heuristics in a stage at a time, we will be able to solve all stages in the time of *m* heuristics plus the administrative time of splitting and joining the solutions. With non-parallel heuristics, this is the fastest we can solve this problem. This speedup is a good one. We can turn a quadratic problem into a linear one, on the number of heuristics.
The number of subproblems (decisions) into which we divide a Steiner problem is $m = n - n_X$. The complexity of each subproblem depends on the heuristic used. In our initial tests we used a very simple heuristic that we called `DijkstraPlusPrune`. It consists of picking randomly an initial node and determining the Dijkstra tree from that node. It is a solution because it is a tree that covers all the Terminal nodes, since the Dijkstra tree covers all nodes in a connected graph. After finding the tree, we proceed pruning all non terminal nodes with degree one, that means, unnecessary nodes for the connectiveness of the terminal nodes. The cost of determining the Dijkstra tree is $\mathbf{O}(n^2)$.
Since every decision takes $2n_X$ executions of the heuristic, we can determine that the order of each decision is $\mathbf{O}(n^3)$. Now we can see that the order of execution of the whole SN metaheuristic is $\mathbf{O}(n^4)$.
Being more precise, the execution order is $\mathbf{O}((n-n_X)^2 n^2)$ that is equivalent to $\mathbf{O}(n^4)$ when $n >> n_X$. This is the most general case in STP resolution. We can also see that if $n \approx n_X$ we will not get an $\mathbf{O}(n^4)$ execution time but it will approximate to $\mathbf{O}(n^2)$. If $n = n_X$ our Steiner Problems turns into **SPG**(*V*, *E*, *w*, *V*) that is equivalent to Dijkstra(*V*, *E*, *w*), whose resolution time is $\mathbf{O}(n^2)$.

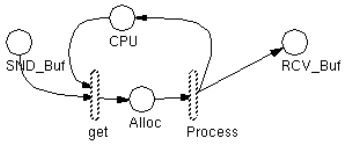
The model

We can see clearly that solution method problem falls within the class of *master-slave* parallel programs, thus, we will apply the general procedure for building a network that represents the system. As we saw on the taxonomy analysis for this kind of problems, they can be represented by the junction of two basic networks, one for the master and other for the slaves as follows:

First model

Master *Slave*

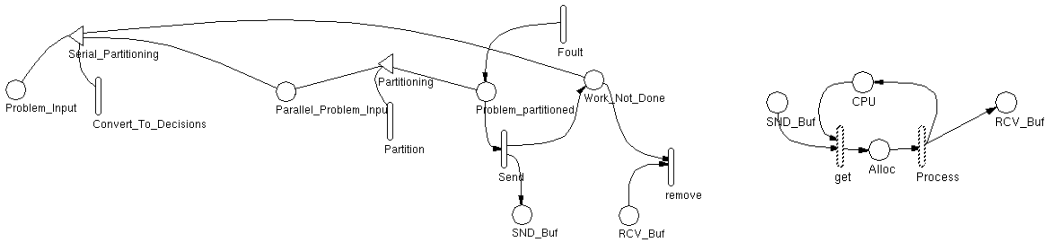




We can see that the resolution of the whole problem consists in $m=|V \setminus X|$ decisions, one for each non terminal node. There exists an implicit synchronization after each of this decisions while the master determines the best decision. This means that there is an outer iterative stage at the master that controls the serial completion of decisions. A more accurate network that models the master process follows:

Second model

Master *Slave*



We will use this second experiment to test how important is to be able to model the synchronization in this particular problem. We will try to predict parallel system performance without modifying the general network for this kind of problems, initializing it with $m(m\text{-}1)$ heuristics to solve. With this general network we will miss m synchronizations amongst all processors. We will try evaluate for this particular case expressiveness vs. simplicity.
It is worth mentioning that this might be interesting considering when the number of decisions outnumbers the CPUs. Lets say that we have k CPUs and m non-terminal nodes. If $k>>m$ we can always address all heuristics needed to take each decision in parallel, but no other decision can be taken until we have taken the decision, that is, $k\text{-}2m$ CPUs will be idle because we can not start solving further heuristics until we take a decision. In this case, if we model the system with our first model, not considering the blocking between decisions, we will estimate a performance that exceeds the real performance of the algorithm. We will study a case in which $k<<m$.

First experiment: *Heterogeneous, single OS, two machines cluster*

The first of our experiments will be run in parallel in two CPUs, the simplest parallel non-SMP scenario. Both systems will be booted on Windows ME (4.90.3000) and the virtual machine used is Sun Microsystem's Java 2 Standard Edition 1.4.0 (build 1.4.0-b92).
Trying to get a clearer graphic representation of the network, we shall assemble a single network with the master and the two slaves instead of using the join capability of UltraSAN. For bigger networks this is recommendable, even though there should be as many slaves as "classes" of CPUs, and as many tokens into the `CPU` places as processors in each one of the classes.
A couple of minor changes were made to the suggested network, mainly due to the iterative

nature of the problem. First of all, we considered no faults, and thus, fault activity was removed. A place called `Decisions_Taken` was added to "count" the number of iterations, and thus decisions that have been taken so far. Also a place called `Sync` was added so as to synchronize the movement of tokens "out" of the network to the `Decisions_Taken` place. All this modifications make it possible to model properly the blocking of the different stages of the problem in the network. It is also possible to skip the usage of the `Sync` place and to complicate the logic of the input gates that control the movement of the tokens along the network. We preferred this option because we believe the semantics of the network are much clearer, making it more understandable. It is also important to mention that these places do not increase the state-space of the network and do not make it more complex for a system to solve it.

Another modification was performed on the network: `Problem_Partitioned` place was removed from the network and the tokens are moved directly to their destinations instead of through it. The reason for the removal is not semantical, but to cut down the number of states generated. Considering this place we would need to consider process resolution as tokens are being moved to the `SND_Buf`: slaves might start solving pieces of work while the `SND_Buf` place is being fed. Even though the inclusion of this place makes a theoretically more accurate network, we found it better to remove it as problem generation times would be too small, and prediction would become inaccurate and state-space would grow significantly.
Returning to the first model, the *naive* one, for the two host experiment, we condensed the first model into one complete network for the same reasons.

It is clear to notice that the level of blocking in the second network is smaller than the one of the first network. It is clear that the second network does not model the real execution, but we want to determine how inaccurate it is to apply the model directly without considering the particular interaction details of this problem.

Parameter fitting

We have the layout of the network that models the system. We need now to determine the performance indexes of the timed activities and the number of slaves available for solving heuristics. The distribution functions that need to be defined according to empirical data are the ones associated with the following timed activities: `Partition`, `Fault`, `Send`, `Remove`, `Get` and `Process`. We will model a perfect system, one in which no `Fault` occurs.

Experimental data

In this section we will empirically determine the required parameters to complete the network definition. As we stated before in the taxonomy analysis, we need to determine both hardware performance indexes and problem complexity, specified in some adequate, problem dependent unit.
We based our tests on the B series of the SteinLib [KMV1]. We worked with the first 7 problems of the suite. The following table resumes some relevant characteristics of the problems:

					Heuristics		
	Nodes	Terminals	Edges	Decisions	Theoretical	Solved	
b01	50	9	63	41	1640	1360	82,91%
b02	50	13	63	37	1332	1194	89,61%
b03	50	25	63	25	600	571	95,22%
b04	50	9	100	41	1640	1563	95,30%
b05	50	13	100	37	1332	1308	98,22%
b06	50	25	100	25	600	587	97,83%
b07	75	13	94	62	3782	3252	85,98%

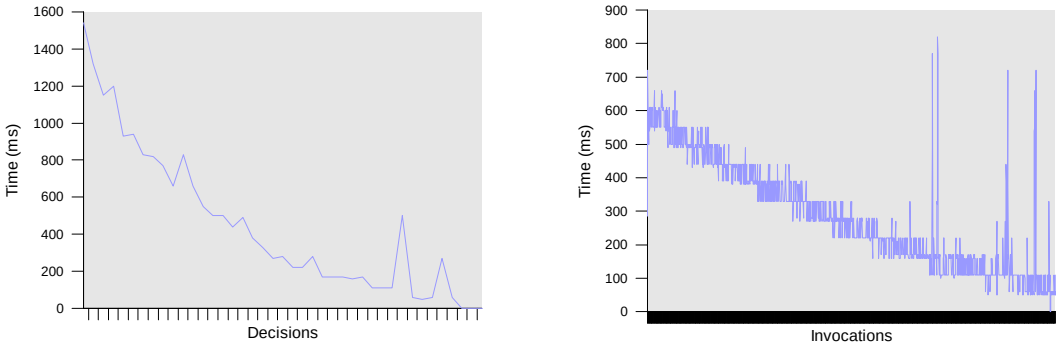
The first three columns show parameters that determine the complexity of the network, parameters that determine directly the size and connectiveness of the graph. The following two columns, Decisions and Theoretical show the complexity of the algorithm measured in the number of decisions that would be taken through the resolution and the number of heuristics that have to be solved in the worst case. The next column, Solved, shows the average number of actual invocations to the heuristics routine in our studies. The difference is explained due to non-connected graphs that are discarded without being considered when exploring the decision space. In the worst case of a fully connected graph, these two figures will be the same.

We performed the first set of our tests on two machines, two similar Celeron systems, one with 1 GHz processor and the other with a 1,1 GHz processor, both of them running Windows 9x OS. The tests consisted of running the master and the slave process locally in each machine three times for each of the seven selected problems. The original codes were slightly modified so as to get timing information. The slave processes were coded so they can measure the time elapsed for each invocation, and the time is printed on the standard output. The master process was modified so it prints on his standard output the time spent partitioning the problem. That allowed us to collect detailed execution data: text files later analyzed. The execution conditions were kept as stable as possible (no other processes were running on the systems), but very high variance was obtained in the measures, suggesting a certain lack of stability in the OS's CPU allocation times.
The following table resumes the information gathered

		b01		b02		b03		b04		b05		b06		b07	
1.1GHz	Total Time	438853	ms	442550	ms	252397	ms	736583	ms	603103	ms	339767	ms	3267740	ms
	Master	443	ms	469	ms	357	ms	661	ms	606	ms	483	ms	1332	ms
	Remote	285	ms	336	ms	383	ms	427	ms	415	ms	509	ms	950	ms
1.0GHz	Total Time	543740	ms	475527	ms	271900	ms	813377	ms	702697	ms	407567	ms	3355377	ms
	Master	520	ms	506	ms	388	ms	707	ms	686	ms	552	ms	1369	ms
	Remote	355	ms	357	ms	421	ms	468	ms	482	ms	593	ms	991	ms

Apart from the data shown in the previous table, there are some numerical aspects that should be taken into consideration. The first thing that has to be considered is that in general, the decision resolution time decreases as the problem resolution takes place. This happens because the problem itself becomes smaller after each decision.
As soon as a decision is taken, one fewer node has to be considered in the next iteration. If the node remains, the graph stays the same, but if the decision consists in removing the node, then the graph that must be considered in the next iteration counts one fewer node and all its incident edges. It is quite reasonable to find this behavior in the general case as the algorithm proceeds pruning the graph until a minimal set of nodes remains, and then only a coverage tree

is saved. Even though there are particular sick-situations where nodes and edges are not removed (i.e. a tree with terminal nodes as leaves), the general case consists of successive smaller graphs, which lead to smaller execution times.
The following graphs plots the data gathered out of a single execution of the **b01** problem. The data itself is not relevant, but the general behavior is. The graph on the left represents the execution times of the problem partitioning at the master process while the graph on the right plots the times thrown at the `RemoteGraphSolver`, slave process's CPU time. We can see that clock resolution is about 50-60 ms. The lack of smoothness on both graphs shows the lack of stability of the OS.



Even though we found that the average on these measured times is suitable for our studies, it is possible to use other functions to estimate problem resolution time expected values every time. In our study variance values are extremely high and are not considered. It might be of interest to use a function of the number of decisions or invocations instead of a constant one. Such function would fit better the gathered data and should be a better model of the reality: as nodes and edges are removed, each heuristic resolution is applied to a smaller graph, thus it's resolution is simpler than the previous one.

We have collected relevant performance values that should suffice to estimate the performance of the parallel system. As suggested, we will use deterministic and exponential distribution functions on our networks to model optimistic and pessimistic executions respectively. We will determine first the optimistic execution times and later, the pessimistic ones.

We will use deterministic functions to estimate TET and MES for this problem. As was stated before, the tool we are using does not provide all the simulators we would like for working with deterministic functions in our particular networks[34]. We shall now study the network analytically.
First of all, we will study how tokens are moved from `Parallel_Problem_Input` place to `Decisions_taken`. We can isolate this study because, until all tokens are removed from this sub-network, `Sync` place will hold a token, and the activity `Convert_To_Decisions` will be paused. When all work is done, the sub-network will be "re-set", a new token will be set at `Decisions_Taken` place and the token will be removed from the `Sync` place.
Lets assume that we have *n* tokens at `Parallel_Problem_Input` place. `Partition` activity becomes enabled and after t_{master} *ms* all tokens are doubled and moved at once to `SND_Buf` place, modeling all possible decisions. After that, both `get1` and `get2` activities are enabled,

34 Maybe we still do not know an equivalent way to compute our results within the tool.

`CPU1` and `CPU2` get allocated and then, `Process1` and `Process2` get enabled. As soon as these activities are finished ($t_{remote1}$ *ms* and $t_{remote2}$ *ms* respectively) CPUs are de-allocated and tokens moved to `RCV_Buf` place. From that moment on, both CPUs will compete consuming tokens from `SND_Buf` place and placing them `RCV_Buf` place in at a speed of $\frac{1000}{t_{remote1}}$ and $\frac{1000}{t_{remote2}}$ tokens per second respectively. As the token consumption is taken place simultaneously, all tokens will be removed approximately after $\frac{t_{remote1} \times t_{remote2} \times n}{(t_{remote1} + t_{remote2}) \times 1000}$ seconds. Considering the master partitioning time, we can state that all tokens are removed in $\frac{t_{master}}{1000} + \frac{t_{remote1} \times t_{remote2} \times n}{(t_{remote1} + t_{remote2}) \times 1000}$ seconds[35].

We should modify this optimistic estimation: it is indeed pessimistic. We realized before that this is the worst case, in which all decisions lead to connected graphs. As we observed in the problems we studied, there are generally problems that are discarded at the master and never solved remotely, thus, the number of tokens solved are smaller than *n*, and it is given by a *factor*. The following formula could be considered and optimistic-average-case estimation:

$$\frac{t_{master}}{1000} + \frac{t_{remote1} \times t_{remote2} \times n \times factor}{(t_{remote1} + t_{remote2}) \times 1000} \text{ seconds}$$

Now we know how fast tokens are removed from `Parallel_Problem_Input` place. Every time *n* tokens are placed in `Parallel_Problem_Input` place, a token is removed from `Problem_Input` place. The following formula predicts optimistically the time spent solving the network

$$n \times \frac{t_{master}}{1000} + \sum_{i=1}^{n} \frac{t_{remote1} \times t_{remote2} \times i \times factor}{(t_{remote1} + t_{remote2}) \times 1000} \text{ seconds}$$

or

$$n \times \frac{t_{master}}{1000} + \frac{t_{remote1} \times t_{remote2} \times factor}{(t_{remote1} + t_{remote2}) \times 1000} \times \frac{n(n-1)}{2} \text{ seconds}$$

From this formula, we can see that the speed of the master only affects the linear component of the equation. The n^2 component is only driven by the remote processing speed.
The following table shows the numerical results:

35 This is not absolutely true. The last token can not be partitioned between both CPUs. Only one of them would get the allocation, and thus the last piece of the resolution would happen only at the processing speed of that CPU instead of the sum of both speeds. We accept this little extra error that simplifies the analytical study.

MES	Optimistic (s)	
	Worst-case	Average-case
b01	277	233
b02	248	224
b03	129	124
b04	393	376
b05	320	314
b06	176	173
b07	1917	1660

The difference between both columns is problem dependent, but we can see that the "more connected" the graph is, the closer the average case is to the worst case. If a deeper study is done on this particular problem, the average case should be related better to the worst case with some theoretical background. Maybe residual connectiveness reliability values should be used, as they give a good index for graph discarding.
The figures shown on the table are the TET for the optimistic approach using the detailed network. We will base our estimation for the MES on the previous results. The unit will be the number of decisions taken per unit of time. For that average we have estimated the time in which the whole problem is solved and we do know the number of decisions taken for both the worst and the average case, thus, we can simply determine the worst and average MES for this problem. The following table shows the numerical results:

TET	Optimistic (decisions/s)	
	Worst-case	Average-case
b01	5,914	5,835
b02	5,373	5,330
b03	4,641	4,625
b04	4,170	4,156
b05	4,168	4,163
b06	3,400	3,395
b07	1,973	1,959

When we first stared at the data, it seemed something wrong: why the number of decisions per second figure is bigger on the worst case than in the average case? That is because the partitioning time is independent from the number of graphs discarded due to lack of connectiveness. This factor makes the average case "slower" or, in other words, the time spent at the master "affects" more fewer decisions.
We also suggested the idea of using a *naive* approach for the model. Let us now estimate the TET and MES for that approach. It is straightforward to see that the *naive* network is a sub-network of the detailed network, and thus, the equation that describes how long it takes for the system to solve the problem is:

$$\frac{t_{remote1} \times t_{remote2} \times factor}{(t_{remote1} + t_{remote2}) \times 1000} \times \frac{n(n-1)}{2} \quad \text{seconds}$$

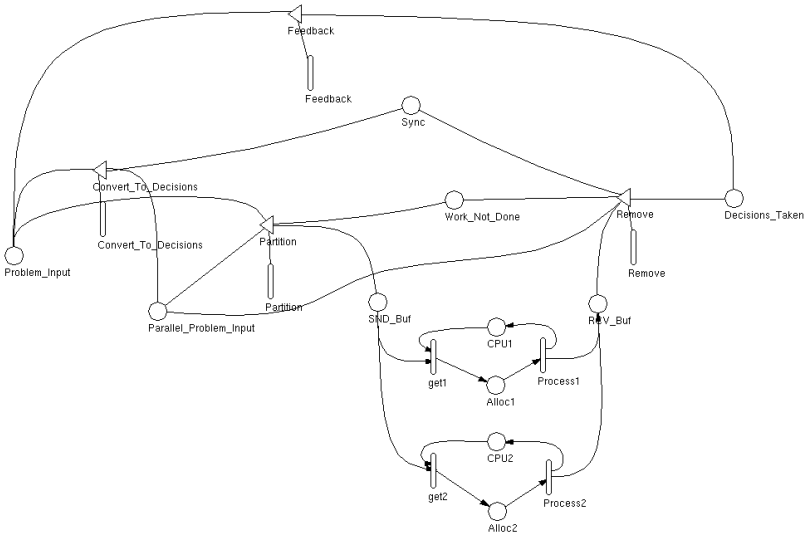
We already know the number of tokens (decisions) that need to be consumed (taken) for each of the b0x problems. The following table resumes the results of applying the previous formula to each problem:

TET	Optimistic (s)	
	Worst-case	Average-case
b01	259	215
b02	231	207
b03	120	115
b04	366	349
b05	297	292
b06	164	161
b07	1834	1577

Thus, the MES Table follows.

MES	Optimistic (decic./s)	
	Worst-case	Average-case
b01	6,328	6,328
b02	5,778	5,778
b03	4,985	4,985
b04	4,479	4,479
b05	4,482	4,482
b06	3,650	3,650
b07	2,062	2,062

It should be quite straightforward to see that the MES is the same for both the average and worst case: no master partitioning time is considered, thus, the only relevant time here is the processing one.
Now that we have estimated the optimistic behavior, not only on the detailed approach but also the *naive* one, we shall study the pessimistic one. This study presents some details that are worth mentioning. On our previous study we were able to estimate our performance indicators due to the particularly small size of the network: 751 states. The previous models lead to networks with one order of magnitude of states more. This networks become slower to estimate and other approaches were used. The kind of study based on successive terminating simulations made the estimation very time consuming and error prone. This time a modification of the network was performed, so it continuously loops. The modified network follows:



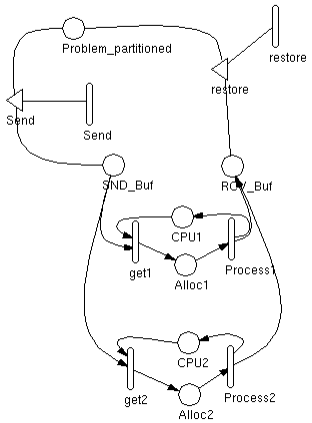
This network avoids the absorbing marking produced when all tokens are moved to `Decisions_Taken` place in the original network. We added a known time to the activity `Feedback`, associated to the transition from the "last" to the "first" state. This transition let us study how long, in average, the network will be performing the `Feedback` activity, and thus,

how long it will be doing the processing we need to estimate. In this case, our tool gives us very good assistance, provided that it calculates our performance variables on the *Direct Steady State Solver* with a good level of accuracy.
We defined a performance variable, `probability`, as an impulse reward. Lets call T_c to the total cycle time, T_p to the problem time and T_f to the feedback time. The total cycle time is the sum of both problem time and the feedback time: $T_c = T_p + T_f$. The value estimated at the simulation is $\frac{1}{T_c}$ thus, we can calculate T_p as $\frac{1}{probability} - T_f$.
The following table resumes the pessimistic estimations for each network, both the reward variable estimated and our estimation for the T_p.

	Pessimistic - Worst Case	
	Probability (1/s)	Pessimistic(s) TET
b01	3,65948E-03	272
b02	3,73238E-03	267
b03	6,90997E-03	144
b04	2,37100E-03	421
b05	2,89707E-03	344
b06	5,06343E-03	196
b07	4,97679E-04	2008

	Pessimistic – Average Case	
	Probability (1/s)	Pessimistic(s) TET
b01	4,01148E-03	248
b02	4,16707E-03	239
b03	7,35991E-03	135
b04	2,50383E-03	398
b05	2,98603E-03	334
b06	5,26087E-03	189
b07	5,77176E-04	1732

In the same way that we estimated the optimistic performance indexes with both the detailed and the *naive* Petri network, we shall now present the results of studying exponential distribution functions on the modified *naive* network. The modifications performed on the network have the same nature than the ones performed on the detailed network: avoid the absorbing marking when all tokens are located at the `RCV_Buf` place. A feedback activity is added so as to re-cycle the tokens from the absorbent marking to the initial marking in a known time. The following network represents the modified network:



We ran the simulation for all the tests. The following table presents the data gathered for both, the worst and the average cases:

	Worst Case - *naive*	
	Probability (1/ms)	Pessimistic(s)
b01	3,84233E-03	259
b02	4,32153E-03	230
b03	8,27571E-03	120
b04	2,72310E-03	366
b05	3,35645E-03	297
b06	6,06341E-03	164
b07	5,44321E-04	1836

	Average Case - *naive*	
	Probability (1/ms)	Pessimistic(s)
b01	4,63279E-03	215
b02	4,82058E-03	206
b03	8,69527E-03	114
b04	2,85716E-03	349
b05	3,41798E-03	292
b06	6,19746E-03	160
b07	6,33134E-04	1578

Lets now present in condensed tables all the estimations performed for the TET for the detailed and naive networks:

TET (s)			b01	b02	b03	b04	b05	b06	b07
Detailed	Optimistic	Average	233	224	124	376	314	173	1660
		Worst	277	248	129	393	320	176	1917
	Pessimistic	Average	248	239	135	398	334	189	1732
		Worst	272	267	144	421	344	196	2008

TET (s)			b01	b02	b03	b04	b05	b06	b07
Naive	Optimistic	Average	215	207	115	349	292	161	1577
		Worst	259	231	120	366	297	164	1834
	Pessimistic	Average	215	206	114	349	292	160	1578
		Worst	259	230	120	366	297	164	1836

We also present here the MES estimated:

MES (s)			b01	b02	b03	b04	b05	b06	b07
Detailed	Optimistic	Average	5,84	5,33	4,63	4,16	4,16	3,40	1,96
		Worst	5,91	5,37	4,64	4,17	4,17	3,40	1,97
	Pessimistic	Average	5,48	5,00	4,23	3,93	3,92	3,11	1,88
		Worst	6,03	4,99	4,17	3,90	3,87	3,06	0,89

MES (s)			b01	b02	b03	b04	b05	b06	b07
Naive	Optimistic	Average	6,33	5,78	4,99	4,48	4,48	3,65	2,06
		Worst	6,33	5,78	4,99	4,48	4,48	3,65	2,06
	Pessimistic	Average	6,33	5,80	5,01	4,48	4,48	3,67	2,06
		Worst	6,33	5,79	5,00	4,48	4,48	3,66	2,06

Observing the last table we see that the four calculations determine the same MES estimated value for each of the different predictions. This is due to the lack of blocking in the whole execution. The other *naive* estimations differ in values as they consider different number of tokens.
The following table presents the experimental data collected from the parallel resolution of the problem:

	b01	b02	b03	b04	b05	b06	b07
TET (s)	267	250	135	371	362	197	1781
MES (s)	5,09	4,76	4,20	4,13	3,60	2,96	1,82

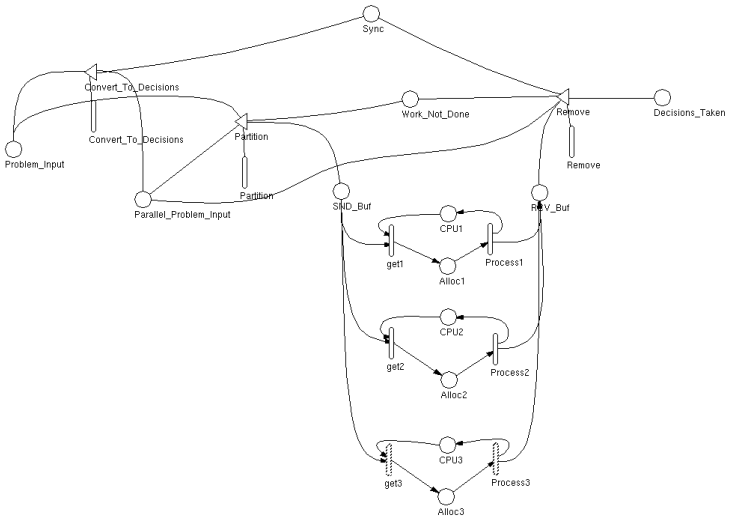
Analyzing the numerical data gathered we can see that the *naive* approach is almost as good as the detailed one, and is always a lower bound for the detailed one. It is good to note that the naive estimation is within the same order of magnitude of time than both the detailed estimation and the numerical solution. We believe that it is also an acceptable performance estimation and in the event of too complex numerical simulations, it could be used.

Second experiment: *Heterogeneous, two OSs, five machines cluster*

We performed the second set of our tests on five machines configured as follows: 1 Pentium III 933Mhz, 512MB RAM, Linux 2.4.18; 3 Pentium II 400MHz, 256 MB RAM; 1 Pentium II 400MHz, 256 MB RAM Windows NT 4.0. The virtual machine used is Sun Microsystem's Java 2 Standard Edition 1.4.0 (build 1.4.0-b92) on windows and linux systems.
The tests consisted in running the master and the slave process locally in each machine three times for each of the seven selected problems. The execution conditions were kept as stable as possible without modifying the standard system configuration excessively.
The following table resumes the information gathered.

		b01	b02	b03	b04	b05	b06	b07
P III 933 Mhz – Linux – 512 MB RAM	Total Time	573578 ms	518439 ms	282640 ms	862685 ms	735778 ms	435485 ms	3662427 ms
	Remote	373 ms	385 ms	433 ms	495 ms	509 ms	634 ms	1082 ms
	Master	585 ms	551 ms	432 ms	792 ms	759 ms	622 ms	1580 ms
P II 400 Mhz – Windows – 256 MB RAM	Total Time	1032942 ms	941410 ms	527672 ms	1594033 ms	1397483 ms	761562 ms	7227974 ms
	Remote	678 ms	715 ms	822 ms	938 ms	982 ms	1119 ms	2141 ms
	Master	1007 ms	985 ms	772 ms	1418 ms	1429 ms	1090 ms	3135 ms
P II 400 Mhz – Linux – 256 MB RAM	Total Time	1214830 ms	1012375 ms	612495 ms	1894883 ms	1595808 ms	821483 ms	7937792 ms
	Remote	792 ms	799 ms	953 ms	1086 ms	1115 ms	1207 ms	2312 ms
	Master	1199 ms	1149 ms	961 ms	1788 ms	1689 ms	1262 ms	3570 ms

We only had to introduce a modification in the Petri Network that models the three different classes of equivalence of CPUs present in this problem, the resulting network follows:



The number of tokens in the CPU places will not be one as in the previous network, as there is one CPU class with three elements. Once again, it could be possible to compose the network using three smaller networks, one for each CPU class and use the joining and replication capability of the tool, but we preferred a simpler construction for this size of network. For the complexity of this network, and considering that we can numerically simulate the detailed network, we will not perform the naive analysis, on this occasion.

We have collected relevant performance values that should suffice to estimate the performance of the parallel system. As suggested, we will use deterministic and exponential distribution functions on our networks to model optimistic and pessimistic executions respectively. We will determine first the optimistic execution times and later, the pessimistic ones.

We will first use deterministic functions to estimate TET and MES for this problem so as to get our optimistic estimations. First of all, we will study how tokens are moved from `Parallel_Problem_Input` place to `Decisions_taken`. We can isolate this study because, until all tokens are removed from this sub-network, `Sync` place will hold a token, and the activity `Convert_To_Decisions` will be paused. When all the work is done, the sub-network will be "re-set", a new token will be set at `Decisions_Taken` place and the token will be removed from the `Sync` place.
Lets assume that we have *n* tokens at `Parallel_Problem_Input` place. `Partition` activity becomes enabled and after t_{master} *ms* all tokens are doubled and moved at once to `SND_Buf` place, modeling all possible decisions. After that, all `get`*i* activities are enabled, `CPU`*i* gets allocated and then, `Process`*i* get enabled. As soon as these activities are finished ($t_{remote1}$ *ms*, $t_{remote2}$ *ms* and $t_{remote3}$ *ms* respectively) CPUs are de-allocated and tokens moved to `RCV_Buf` place. From that moment on, all CPUs will compete consuming tokens from `SND_Buf` place and placing them `RCV_Buf` place in at a speed of $\frac{1000}{t_{remote1}}$, $\frac{1000}{t_{remote2}}$ and $\frac{3000}{t_{remote3}}$ tokens per second respectively. As the token consumption is taken place simultaneously, all tokens will be removed approximately after $\frac{t_{remote1}\times t_{remote2}\times t_{remote3}\times n}{(t_{remote2}\times t_{remote3}+t_{remote1}\times t_{remote3}+3\times t_{remote1}\times t_{remote2})\times 1000}$ seconds. Considering the master partitioning time, we can state that all tokens are removed in $\frac{t_{master}}{1000}+\frac{t_{remote1}\times t_{remote2}\times t_{remote3}\times n}{(t_{remote2}\times t_{remote3}+t_{remote1}\times t_{remote3}+3\times t_{remote1}\times t_{remote2})\times 1000}$ seconds[36].

Now we know how fast tokens are removed from `Parallel_Problem_Input` place. Every time *n* tokens are placed in `Parallel_Problem_Input` place, a token is removed from `Problem_Input` place. The following formula predicts optimistically the time spent solving the network

$$n\times\frac{t_{master}}{1000}+\sum_{i=1}^{n}\frac{t_{remote1}\times t_{remote2}\times t_{remote3}\times n}{(t_{remote2}\times t_{remote3}+t_{remote1}\times t_{remote3}+3\times t_{remote1}\times t_{remote2})\times 1000}\ \text{seconds}$$

or

$$n\times\frac{t_{master}}{1000}+\frac{t_{remote1}\times t_{remote2}\times t_{remote3}\times n}{(t_{remote2}\times t_{remote3}+t_{remote1}\times t_{remote3}+3\times t_{remote1}\times t_{remote2})\times 1000}\times\frac{n(n-1)}{2}\ \text{seconds}$$

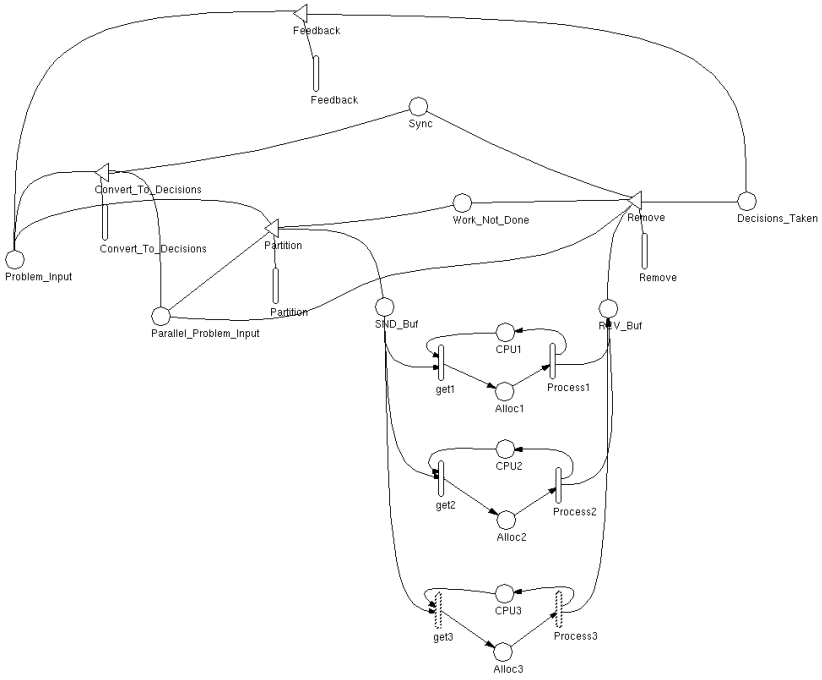
From this formula, we can see that the speed of the master only affects the linear component of the equation. The n^2 component is only driven by the remote processing speed.

36 This is not absolutely true. The last token can not be partitioned between both CPUs. Only one of them would get the allocation, and thus the last piece of the resolution would happen only at the processing speed of that CPU instead of the sum of both speeds. We accept this little extra error that simplifies the analytical study.

The following table shows the numerical results:

	Optimistic	
	TET (s)	MES
b01	122	13,44
b02	104	12,81
b03	53	11,32
b04	165	9,94
b05	140	9,51
b06	75	8,00
b07	790	4,79

Now that we have estimated the optimistic behavior, we shall study the pessimistic one. As it is suggested in the taxonomy analysis, we modified the network so it continuously loops avoiding the absorbent configuration that arises after the processing is done. The modified network follows:



This network avoids the absorbing marking produced when all tokens are moved to `Decisions_Taken` place in the original network. We added a known time to the activity `Feedback`, associated to the transition from the *last* to the *first* state. This transition let us study how long, in average, the network will be performing the `Feedback` activity, and thus, how long it will be doing the processing we need to estimate. In this case, our tool gives us a very good assistance, provided that it calculates our performance variables on the *Direct Steady State Solver* with a good level of accuracy.

We defined a performance variable, `probability`, as an impulse reward. Lets call T_c to the total cycle time, T_p to the problem time and T_f to the feedback time. The total cycle time is the sum of both problem time and the feedback time: $T_c = T_p + T_f$. The value estimated at the

simulation is $\frac{1}{T_c}$ thus, we can calculate T_p as $\frac{1}{probability} - T_f$.

The following table resumes the pessimistic estimations for each network, both the reward variable estimated and our estimation for the T_p.

	Pessimistic	
	TET (s)	MES
b01	345	4,75
b02	288	4,63
b03	170	3,53
b04	469	3,50
b05	402	3,31
b06	229	2,62
b07	2134	1,77

The following tables present a comparison that collects our pessimistic and optimistic estimations together with experimental data obtained on experiments. For our experiments we present the minimum, maximum and average values. The first table presents the TET and the second the MES.

		b01	b02	b03	b04	b05	b06	b07
TET (s)	Pessimistic	345	288	170	469	402	229	2134
	Max	250	207	127	342	306	193	1399
	Average	232	198	120	329	281	187	1360
	Min	218	191	114	309	257	184	1332
	Optimistic	122	104	53	165	140	75	790

		b01	b02	b03	b04	b05	b06	b07
MES	Pessimistic	4,75	4,63	3,53	3,5	3,31	2,62	1,77
	Max	6,55	6,44	4,71	4,79	4,35	3,11	2,70
	Average	7,06	6,71	5,01	4,99	4,73	3,21	2,78
	Min	7,52	6,96	5,27	5,31	5,19	3,27	2,84
	Optimistic	13,44	12,81	11,32	9,94	9,51	8	4,79

We can see that in this case, we obtain a speedup with 5 slower systems than with 2 fast ones if we compare the two experiments.

5.4 - Annex on single CPU multitasking observations

One of the tools developed is the “primos” program. The program finds the prime numbers within a given interval. Determining if a number is prime or not is a simple mathematical problem that has algorithmically been solved since 230 BC by Eratostenes, but there is no equation that can be used to determine if the given number is a prime number or not: it has to be tested. Isaac Asimov wrote about this problem in his essay “Prime quality” in 1966 [ASI1]

and depicted some ways to discard numbers that cannot be prime ones, but there is no known way better than testing. Current technology has developed highly sophisticated techniques to help discarding numbers that are not prime with fewer operations that simple brut-force testing. In our case we want to load CPU, so we did not care about optimization aspects.

The algorithm that we choose to determine if a number is prime is not only simple but inefficient: try all numbers smaller than its half to see if someone is a divisor and is different from number one. When a prime *n* is tested, $\frac{n}{2}$ integer divisions have to be done, which is the worst case. Whenever a number that is not prime is tested, fewer divisions are done. If we try to determine the prime numbers in a given interval $[x_0, x_n]$, $x_n \frac{(x_n-1)}{4} - x_0 \frac{(x_0-1)}{4}$ operations need to be done in the worst case. Here is an upper bound to the complexity of the problem.

We want to distribute the task of finding the prime numbers of an interval among different processes that can be run on different processors or on different processes instances on the same system regardless the number of processors available. We can achieve this dividing the interval in smaller intervals, solving each subinterval and joining the results.

We did it applying bipartition. The first code simply divided the interval in two *by the middle*. It works fine, but the work load on each interval is different, thus the CPU time needed to solve each part is different. We obtained speedups, but there was a period of time in which there was only one process running and the rest of the CPUs waiting for its result. We tried to find a partitioning that divides the problems in two pieces with approximate complexity.

As we stated before, the worst-case number of divisions that have to be done in a given interval $[x_0, x_n]$ is given by $ops(x_0, x_n) = x_n \frac{(x_n-1)}{4} - x_0 \frac{(x_0-1)}{4}$. We want to find x_j belonging to the interval $[x_0, x_n]$ that verifies $\alpha\, ops(x_0, x_j) \approx (1-\alpha)\, ops(x_j, x_n)$. We introduced α so as to have a control in the way we partition the interval. Choosing $\alpha = \frac{1}{2}$ we are partitioning the interval in two subintervals with approximate worst-case complexity. Solving the equation we found that $x_j = \frac{(1+\sqrt{1-4(\alpha(x_0 - x_0^2 + x_n^2 - 1) - x_n^2 + x_n)})}{2}$ is a proper value for our purpose, being x_j a closer integer to the real result of the equation.

We implemented a small C – PVM application that receives the interval, α and the worst case number of operations that can be done by a single process without spawning child processes to solve subintervals.

Our first experiment was to determine the overhead of context switches in the operating system. We tested the prime numbers in the interval $[1,200000]$ with a single system and using $\alpha = \frac{1}{2}$.

Our test consisted in spawning the same problem, for the same interval, but with a different parameter for the partitioning. Successive executions shrinks the maximum number of allowed operations so each child process has to be fragmented one step further. The value for the parameter, named Complexity as it represents the maximum subproblem number of operations allowed for a single process, is calculated so as to reach the specific number of partitions desired. The following table shows the times for different execution times on Linux systems

running SuSE 6.3:

		Celeron 533MHz, 192 MB RAM		Pentium 166MHz, 64 MB RAM	
Complexity	Number of processes	Time	Overhead per process	Time	Overhead per process
9999950000	1	58,3	0,00	284,33	0,00
4999975000	3	59,0	0,22	284,67	0,11
2499987500	9	59,0	0,07	284,67	0,04
1249993750	17	59,0	0,04	285,33	0,06
624996875	33	60,0	0,05	287,33	0,09
312498438	65	60,0	0,03	287	0,04
156249219	129	61,3	0,02	288,67	0,03
78124610	257	62,3	0,02	293,33	0,04
39062305	513	67,7	0,02		

The first column shows the parameter passed to the program as the worst case number of operations allowed to perform without dividing, the second column is the number of processes involved in the solution. We measured the time elapsed in seconds and averaged three runs for each experiment to minimize OS and other tasks interference. We calculated the per-process overhead as the difference in time divided by the number of processes. We see that this parameter stabilizes as the number of processes grows.
The system configuration imposes some limits to the number of processes that can be run on a single system. The complexity could not be subdivided further because deadlock situations or hangs arise randomly.
We can also see that under controlled conditions, the overhead we have to pay for excessive CPU allocation can be predictable (in our case 0,02 or 0,04 ms per process) and can be controlled. For scenarios in which the number of processes spawned exceeds the number of available CPUs we can assume quite straightforwardly that we can assign more than one process to a single CPU and we can expect pretty fair allocation times for each process and model it as different CPUs with the adequate fraction of the original CPU power.

6 - Conclusions & future work

The present document summarizes almost three years of part-time research in the field of parallelism, that begun at the CeCal and finished at the InCo. The main objective of this work is the applied theoretical performance evaluation of large grain parallelism on loosely coupled multicomputers, with private memory, bonded with high speed networks.
The investigation started up with a research of parallel performance evaluation tools and methods that help designing and constructing a parallel cluster and found that there were no complete performance evaluation methods available that help designing a parallel cluster. The objective then moved into the construction of the theoretical bases and models that help achieving objective performance estimations of applications running on particular hardware configuration with the purpose of helping the designer of the parallel machine determining the best configuration.
The mathematical tool selected are the Stochastic Petri networks and the design and simulation tool used was the UltraSAN.
On the previous chapters we introduced the grounds for this analysis and work and also presented the model templates that can be applied to almost any parallel problem solved using parallel hardware. It is important to keep in mind that the target of this study is the large grain parallelism, not the fine grain one. Factors like network speed, processor speed and family, amount of memory, etc. are collapsed within one figure that represents each processor speed. This decision seemed strange at the beginning even to us, but as it was seen during RC5 analysis, the level of detail corresponds with the size of the grain of the parallelism addressed. In our case, we are modeling resolution-wide parameters and large grain parallelism. We have to make an abstraction of each piece of the parallel machine performance, thus many parameters that affect individual node performance are collapsed into a single figure that represent each node's processing power. This imposes some constraints on the level of detail of each processing node and will impose some limits to the problems where we can use the templates. There is no direct way to model the memory access speed, number of bits in the data-path of the PCI bus, etc. Other models ought to be used for this analysis. All this kind of details collapse within a single figure. Our models does not help deciding if it is better to have faster RAM, bigger L2 cache etc. for achieving that single figure, but they compare the effective system performance for the particular problem and the interaction the multiple systems addressing in parallel a task. Particular system details should be analyzed by other means.
After presenting the model templates we applied them in two specific example problems so as to show the way they can be used to predict performance estimators TET and MES The two applications studied were correctly estimated in standard and non standard parallel environments, both with homogeneous and heterogeneous platform characteristics. Even though the size of the problems and the number of systems were relatively small, we were able to gather good predictions for our performance estimators. We modeled the execution using both optimistic and pessimistic approaches, modeling individual resolution times with deterministic and exponential distribution functions respectively. We also found that the general assumption of optimistic behavior associated to deterministic distribution functions and the pessimistic behavior associated with exponential distribution functions is also valid in this models, as we were able to bound the real execution measures with the optimistic and pessimistic estimations. We believe that it might be worth continuing the evaluation of other

distribution functions that can fit better the real execution.
We found several practical problems in the process of solving built models regarding to the Petri network resolution. The main issue was the space of states. Many of the solvers need to generate all possible network configurations before they really solve the network. The complexity of this problem can grow considerably as the number of combinations and options grow. In our examples[37] we had resolution times of several minutes, which generates some problems when the system we model consists of hundreds of nodes. This is the reason why multiple details collapsed within a single figure. We believe that our model templates are detailed enough to capture problem logic and hardware performance but also controls the complexity of the numerical resolution of the generated Petri networks.

It is important to note that the models presented themselves do not build a parallel machine for a certain purpose but help the designer deciding benefits and drawbacks of decisions taken by means of estimation of relevant performance descriptors.
The intended application scenario for these models are small companies or research groups that build their own parallel machines for solving particular problems. In this environments the tools will prove useful helping designers either determining that existing hardware is enough for performance requirements or for justifying investment on newer hardware. This models can also help research groups explore convenience of different algorithms for solving determined problem on certain hardware. This model templates provide means for predicting performance estimators for different algorithms solving the same problem on the same hardware, that can help identifying the best algorithm that can be run on specific hardware to solve a problem when more than one algorithm is available for solving a problem. This is also useful for early algorithm comparisons that may complement complexity analysis as it comprises also blocking and other execution events that slows down execution.
We believe that this theoretical result should be the basis for the development of specific performance analysis tools that can be combined with existing Petri networks. We believe that the next logical step for this research is the construction of an automatic tool for the construction of the Petri network. We can see that it is possible to automate the construction of the Petri network that models the particular problem with a Wizard-based interface that collects information from users, automatically applies the templates and generates the Petri net, that is fed to a tool like UltraSAN that is used to estimate TET and MES. This small step automates one step further current tool, as it isolates the user from building the Petri network and using the simulation tool.
Another usage for this model templates is to build an automatic cluster building tool that can decide the best hardware configuration possible for a particular problem, given certain constraints. It is possible to devise a set of rules that defines how to assemble parts so as to build a computer, a network and finally a cluster. This rules shall describe properly parameters like the number of PCI slots, clock rates at which motherboard operates, types of memory, number of ports in switches and so on. Non directly performance related parameters like heat dissipation, volume occupied, power consumption and cost of the equipment must be considered. It should be also possible to describe rules for joining this parts in a way that only the right number and type of processors are used, the proper media is selected for network cards, the switches are dimensioned for the right number of nodes, etc. Then we should provide “validation” rules that help checking factors like heat dissipation, connectiveness of the solution, etc.: it is not valid to have 10 slaves that cannot connect to a master because the

37 The practical examples analyzed differ in up to 3 or 4 orders of magnitude in the number of nodes with cutting edge commodity parallel projects.

number of ports in the switch are not enough. After having the hardware and the problem definition, the automatic Petri network construction and resolution takes place, and then we can predict performance estimators for our system. This sort of cluster building intelligence is fed with all available components that can be found with their associated parameters and is also fed with a problem definition and a set of constraints and it can produce the finite list of clusters that can address the specified problem with the given constraints and the TET and MES associated. If this list is not short enough for complete evaluation, it is possible to think of GRASP like algorithms that can search locally optimal solutions starting from random ones. To be able to do this it is necessary to define the neighborliness concept for clusters and we are done. Other heuristics like Tabou search, simulated annealing are applicable.

Parallelism is a mature and strong area that has multiple industrial applications and is only starting up in the office environment. Multi-terabytes office environments are predicted for the years to come and software being able to search, index, retrieve and in general process that amount of information will benefit from parallelism. Future parallel environment will differ from current ones and we believe this sort of coarse parallel analysis and tools will become desktop tools for system administrators.

7 - Bibliography

[ABC1] M. Ajmone Marsan, G. Balbo, G. Conte – "Performance Models of Multiprocessor Systems" – MIT Press – ISBN 0-262-01093-3 – 1986

[ASC1] http://www.sandia.gov/ASCI/ (20/9/2002)

[ASI1] Isaac Asimov - "The left hand of the electron - The solar system and back - From earth to heaven" – Alianza Editorial Madrid – ISBN 84-206-1653-2 – 1972

[BEO1] http://www.beowulf.org (20/9/2002)

[BEO2] Bewoulf How-to http://www.canonical.org/~~kragen/beowulf-faq.txt (20/9/2002)

[BEO3] D. Becker, T. Sterling, D. Savarese, J. Dorband, U. Ranawak, C. Packer "Beowulf: A Parallel Workstation For Scientific Computation", Proceedings, International Conference on Parallel Processing, 1995, http://www.beowulf.org/papers/ICPP95/icpp95.ps (20/9/2002)

[BEO4] C. Reschke, T. Sterling, D. Ridge, D. Savarese, D. Becker, P. Merkey "A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation", Proceedings, High Performance and Distributed Computing, 1996 http://www.beowulf.org/papers/HPDC96/hpdc96.ps (20/9/2002)

[BEO5] D. Ridge, D. Becker, P. Merkey, T. Sterling, P, Merkey "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs", Proceedings, IEEE Aerospace, 1997, http://www.beowulf.org/papers/AA97/aa97.ps (20/9/2002)

[BUY1] Rajkumar Buyya. "High Performance Cluster Computing". Prentice Hall. ISBN 0-13-013785-5 – 1999

[BRO1] Robert G. Brown. "So, you want to build a Beowulf? Workload profiling and beowulf design". http://www.phy.duke.edu/brahma/profiling.ps (20/9/2002)

[CAF1] Christopher D. Carothers, Richard M. Fujimoto "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms", IEEE Transactions on Parallel and Distributed Systems, Vol 11, No. 3, March 2000 http://dlib2.computer.org/td/books/td2000/pdf/10299.pdf (20/9/2002)

[CTC1] http://www.ctc-hpc.com (20/9/2002)

[ECK1] B. Eckel. "Thinking in JAVA". http://www.eckelobjects.com (20/9/2002)

[HWG1] J. Hill, M. Warren, M. P. Goda. "I'm not going to pay a lot for this supercomputer!" Linux Journal, 1998 http://www.linuxjournal.com/article.php?sid-2392 (20/9/2002)

[INT1] http://developer.intel.com/design/chipsets/440bx (20/9/2002)

[KMV1] T. Koch, A. Martin, S. Voss. "SteinLib: An Updated Library on Steiner Tree Problems in Graphs" Konrad-Zuse-Zentrum für Informationstechnik, Berlin – ftp://ftp.zib.de/pub/zib-publications/reports/ZR-00-37.pdf (20/9/2002)

[LIN1] Christoph Lindemann – "Performance Modelling with Deterministic and Stochastic Petri Nets" – 1998 – ISBN 0 471 97646 6

[MAR1] S. L. Martins, P. M. Pardalos, M. G. C. Resende, C. Ribeiro "Greedy Randomized Adaptative Search Procedures for the Steiner Problem in Graphs" – DIMACS Series in Discrete Mathematics and Theoretical Computer Science 43 (1999), 133-146 – http://www-di.inf.pvc-rio.br/~celso/artigos/gspg.ps (20/9/2002)

[MAR2] S. L. Martins, C. Ribeiro, M. C. Souza "A Parallel GRASP for the Steiner Problem in Graphs" – Workshop on Parallel Algorithms for Irregular Structured Problems (1998), 285-297 – http://www-di.inf.pvc-rio.br/~celso/artigos/par_grasp_steiner.ps (20/9/2002)

[PVM1] http://www.netlib.org/pvm3/ (20/9/2002)

[PVM2] "PVM A Users Guide and Tutorial for Networked Parallel Computing" http://www.netlib.org/pvm3/book/pvm_book.ps (20/9/2002)

[PVM3] M. Fischer, J.Dongarra. "Another Architecture: PVM on Windows 95/NT" http://www.netlib.org/pvm3/win32/nt_paper.ps (20/9/2002)

[QCC1] Francesco Quaglia, Vittorio Cortellessa, Bruno Ciciani "Trade-Off between Sequential and Time Warp-Based Parallel Simulation", IEEE Transactions on Parallel and Distributed Systems, Vol 10, No. 8, August 1999 – http://dlib.computer.org/td/books/td1999/pdf/10781.pdf (20/9/2002)

[RIB1] C. Ribeiro, M. C. de Souza "Improved Tabu Search for the Steiner Problem in Graphs" – Working paper, Catholic University of Rio de Janeiro, Department of Computer Science (1997) – http://citeseer.nj.nec.com/47337.html (27/9/2002)

[ROB1] F. Robledo "Diseño topológico de redes: casos de estudio 'The generalized Steiner Problem' y 'The Steiner 2-Edge-Connected subgraph problem'". Tesis de Maestría en Informática, PEDECIBA 2000. Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay.

[RSA1] http://www.rsasecurity.com/rsalabs/callenges (27/9/2002)

[RSHD] – "WIN32 - RSHD: A BSD compliant RSH Daemon / RSH Service for Microsoft's WIN32 Architecture" – http://www.winrshd.com (27/9/2002)

[RUS1] - Mark Russinovich – "Inside Win2k Scalability Enhancements, part 2" – http://www.winntmag.com/Articles/Content/7597_01.html (27/9/2002)

[SAB1] – Ariel Sabiguero – "Nomenclatura y definiciones básicas de Redes de Petri" – Reporte Técnico Nro. 02-18" – Instituto de Computación – Facultad de Ingeniería – Universidad de la República - 2002

[SW1] J. Salmon, M. S. Warren. "Parallel out-of-core methods for N-body simulation" 8th SIAM Conf. On Parallel Processing for Scientific Computing, Philadelphia, 1997 http://www.cacr.caltech.edu/~johns/pubs/siam97/salmon.pdf (27/9/2002)

[SET1] http://setiathome.ssl.berkeley.edu/index.html (27/9/2002)

[TAN1] Andrew S.Tanenbaum "Distributed operating systems". Prentice Hall. ISBN 0-13-219908-4 – 1995

[TOP1] http://www.netlib.org/benchmark/top500/top500.list.html (27/9/2002)

[URR1] S. Urrutia, I. Loiseau "A New Metaheuristic and its Application to the Steiner Problems in Graphs" – XXI International Conference of the Chilean Computer Science Society (SCCC'01) (7-9/11/2001) Punta Arenas, Chile – http://dlib2.computer.org/conferen/sccc/1396/pdf/13960273.pdf (27/9/2002)

[USAN] - W.H. Sanders – http://www.crhc.uiuc.edu/UltraSAN (27/9/2002)

[WBG1] M. S. Warren, D. J. Becker, M. P. Goda, J. K. Salmon, T. Sterling "Parallel supercomputing with commodity components" Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97) 1997 http://loki-www.lanl.gov/papers/pdpta97/pdpta97.ps (27/9/2002)

[WSB1] M. Warren, J. Salomon, D. Becker, M. Goda, T. Sterling, G. Winckelmans. "Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac" http://loki-www.lanl.gov/papers/sc97/ (27/9/2002)