POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione

Models for SIMD, MIMD and Hybrid Parallel Architectures

Realizzazione di modelli di varie architetture di calcolatori paralleli (di tipo SIMD, MIMD e ibridi) e loro validazione sperimentale

Claudio Gennaro

Advisor: Prof. Giuseppe Serazzi

Ph.D. Thesis - 1995/1998 - Dottorato di Ricerca in Ingegneria Informatica e Automatica

POLITECNICO DI MILANO
Dottorato di Ricerca in Ingegneria Informatica e Automatica

Realizzazione di modelli di varie architetture di calcolatori paralleli (di tipo SIMD, MIMD e ibridi) e loro validazione sperimentale

Tesi di Dottorato di:
Claudio Gennaro

Relatore:
Prof. Giuseppe Serazzi
Tutore:
Prof. Giuseppe Serazzi
Coordinatore del Dottorato:
Prof. Carlo Ghezzi

XI ciclo

Sommario

I supercomputer massivamente paralleli sono utilizzati sempre più frequentemente per soddisfare le esigenze di calcolo ad alte prestazioni sia nel mondo della ricerca sia in quello dell'industria. Le prime architetture parallele apparse sul mercato erano di tipo vettoriale, architetture in cui i processori erano in grado di eseguire in modo parallelo una stessa operazione sui diversi elementi di un vettore. Volendo sfruttare in modo massiccio il grado di parallelismo di un'applicazione, è apparsa evidente la necessità di realizzare architetture parallele di tipo general-purpose, costituite da un elevato numero di processori dotati di memoria locale e interconnessi mediante un'opportuna rete di comunicazione. Allo stato attuale il calcolo parallelo viene realizzato attraverso macchine specializzate per il supercalcolo MIMD (Multiple Instruction Multiple Data) o SIMD (Single Instruction Multiple Data) oppure attraverso reti di workstation. Una nuova famiglia di supercomputer, oggetto di ricerca presso l'ENEA, è composta da sistemi paralleli ibridi. In queste architetture un sistema di supercalcolo MIMD utilizza un computer di tipo SIMD come un insieme di coprocessori per le operazioni di calcolo a virgola mobile più complesse. In questo modo si accoppia la flessibilità dei sistemi MIMD nel trattare diversi problemi, con la potenza di calcolo dei sistemi SIMD. In questo lavoro di ricerca si propone un approccio sistematico per l'analisi modellistica del comportamento di applicazioni parallele, in termini di andamento delle richieste di elaborazione, delle comunicazioni e delle operazioni di I/O, prendendo in considerazione le architetture del tipo MIMD, ibride e cluster di workstation. Dal punto di vista del software si utilizza un modello tipico della maggior parte delle applicazioni parallele. Il comportamento di tali programmi è, infatti, schematizzabile come una sequenza fasi, ognuna delle quali consiste in singoli burst di computazione seguiti da singoli burst di I/O. A sua volta i burst di computazione possono essere ulteriormente scomposti in burst di calcolo (in cui sono coinvolte solo operazioni di calcolo dei singoli processori) e burst di comunicazione (in cui viene coinvolta la rete di interconnessione dei processori). Le comunicazioni tra i processori rappresentano senz'altro un aspetto rilevante nel determinare le prestazioni dei sistemi paralleli. Il modello proposto consente di tener conto di tre elementi importanti presenti nelle comunicazioni:

- le dimensioni dello spazio dei dati utilizzati dall'applicazione: in questo modo si può determinare come scalano le comunicazioni con il numero di processori;
- il livello di contesa sulla rete, che dipende sia dal tipo di hardware di comunicazione (maglia, memoria comune, albero, etc.) sia dal tipo di algoritmo parallelo;

- il numero di processori che si sincronizzano durante le comunicazioni. Può variare da uno (comunicazioni asincrone) fino al numero massimo di processori impiegati dall'applicazione.

L'I/O nei sistemi paralleli costituisce il secondo fattore limitante nelle prestazioni di applicazioni di supercalcolo. Nei sistemi MIMD e nei cluster di workstation viene preso in considerazioni l'I/O verso dispositivi di memoria di massa. Nei sistemi ibridi diventa importante l'I/O utilizzato per la comunicazione fra il sottosistema MIMD e il sottosistema SIMD. Lo studio di quest'ultimo tipo di I/O è importante in quanto la comunicazione fra MIMD e SIMD può diventare il collo di bottiglia di un'applicazione parallela che voglia sfruttare a pieno la potenza della tecnologia ibrida. I modelli proposti permettono di ottenere speedup teorici in termini del numero di dischi disponibili o del numero di nodi SIMD della macchina reale. Sono stati previsti due tipi di configurazione di I/O:

- *sottosistema centralizzato*: in cui l'I/O è gestito da un singolo nodo della macchina parallela attraverso un canale di comunicazione connesso ad un disco RAID o a una board SIMD;

- *sottosistema distribuito*: in questo caso gruppi di nodi della macchina parallela condividono un canale di I/O tipicamente connesso ad un disco.

Oltre al tipo di configurazione dell'hardware di I/O, sono stati presi in considerazione anche le diverse modalità di I/O delle applicazioni parallele:

- I/O sincroni: tutti i processori coinvolti dall'applicazione effettuano contemporaneamente trasferimenti attraverso l'I/O. Si pensi ad esempio, nel caso di I/O verso unità di massa, al checkpoint per visualizzazione e analisi dei dati;

- I/O asincroni: i processori effettuano I/O in momenti diversi. Si pensi al caso della computazione out of core, in cui ogni processore salva i dati in modo indipendente dagli altri allo scopo di liberare spazio nella memoria RAM locale.

Allo scopo di validare i modelli proposti in questa ricerca, due tipi di approcci sono stati utilizzati. In una prima fase è stata realizzata una versione parallela dell'algoritmo MVA per la soluzione di reti di code particolarmente complesse. Essendo note le caratteristiche di tale applicazione è stato possibile stimare i parametri da introdurre nel modello. Lo speedup ottenuto dal modello è stato infine confrontato con quello sperimentale. In una seconda fase sono stati utilizzati gli speedup di alcune applicazioni parallele forniti dalla Scalable I/O Initiative. Poiché i parametri utilizzati dal

modello corrispondono a caratteristiche misurabili di un programma parallelo, è possibile, attraverso tecniche di approssimazione ai minimi quadrati fra lo speedup osservato e quello ottenuto dal modello, inferire i parametri del programma.

POLITECNICO DI MILANO
Dottorato di Ricerca in Ingegneria Informatica e Automatica

Models for SIMD, MIMD and Hybrid Parallel Architectures

Ph.D. Thesis of:
Claudio Gennaro

Advisor:
Prof. Giuseppe Serazzi
Tutor:
Prof. Giuseppe Serazzi
Supervisor of the Ph.D. Program:
Prof. Carlo Ghezzi

XI ciclo

To Floriana

Acknowledgments

I would like to thank my supervisor Prof. Giuseppe Serazzi for the inspiration, guidance, friendship offered throughout my studies.

I feel a special gratitude towards Peter King and Paolo Cremonesi for their important contribution to the work presented in this thesis.

My family and friends contributed greatly to my work with their unending encouragement. In particular, I thank my mother for listening and my girlfriend Floriana for her love and care.

My thanks to my friends: Francesca Alonzo, Fabio Casati, Laura Castanò, Gianfranco Castriotta, Angela Discenza, Pier Luca Lanzi, Maristella Matera, Fabio Violante. They kept my spirits high!

During my stay in Edinburgh in Scotland, I appreciated Mario Antonioletti and Armen Avedisijan for being very friendly and helpful.

The Italian Agency for New technology, Energy and the Environment (ENEA) greatly acknowledged for its financial support.

Contents

Chapter 1

Introduction

The purpose of massively parallel machines is to solve problems in various scientific/engineering domains that would have unacceptable processing times if solved with a traditional single processor machine. The performance analysis has emerged as a key area of research in the field of parallel architectures, ever since the first high performance machines appeared on the market. This appeared ever more true as the trend of the high performance computer moved from using vector and array processors to using multiprocessor architectures. Many factors influence the performance of parallel applications. Hardware or software components, or both can be involved. Examples are: processor speed, processor communication network, I/O architecture, sequential code, communication network contention and synchronization protocols, data partitioning, etc.

The first effort to model the performance of parallel programs is due to Amdahl and the well known homonymous law that relates the number of processors to the bound of the speedup which may be expected by parallel processing [1]. This bound has proved useful in shaping our understanding of parallel system because it strikes a useful balance between simplicity and precision. Several amendments and extensions to Amdahl's law have been proposed, each appropriate for different purposes. Gustafson et al. [23, 25, 24] introduce the concept of scaled speedup as a measure of how well an algorithm scales when the number of processors is multiplied by k. Flat et al. [16] investigate the impact of synchronization and communication overhead on the performance of parallel processors. Eager et al. [13] studied speedup versus efficiency. Wu et al. [47] propose a formal definition of scalability and discuss scalability of cluster systems. Finally, Rosti et al. [42] extends Amdahl's law to three–dimensional space to include processors and disks. However, these studies are limited by the fact that the speedup formulations do not include any explicit parameters that

reference the communication and I/O overhead or the type of hardware used for executing the parallel application.

Other approaches use task graph models in order to represent the inter-task precedence relations within a program, the task execution times, and the times necessary to transfer data or other information from one task to other during execution. In the case of program control structures that can be represented by "series–parallel" task graphs, such as the fork–join structure, methodology to predict the best speedup which can be obtained with such program has been developed. In particular, when the task execution times are deterministic, Coffman et al. [9] used critical path analysis to find the program completion time. If the task execution times are stochastic and the number of processors is infinite, the probability distribution time can be determined by a straightforward but in general very costly computation [28, 3]. General acyclic random graph models are presented in [15, 36, 12]. Gelenbe [18] generalizes the task graph models so that it is possible to take into account the effect of communication times between the tasks.

For more realistic cases where the number of processors is smaller than the number of tasks in a program, a queueing network approach can be used. Approximate models are presented in [8, 19, 11] and an exact model is proposed by Baccelli et al. [2].

Task graphs, as general acyclic graphs, allow the description of sequential or parallel execution, synchronization, and spawning of task. Nevertheless, the multiprocessor systems considered have a generic architecture with a finite number of processors sharing a central memory.

This work proposes a new model for parallel systems with distributed computational and I/O resources when executing parallel applications characterized by cyclic computation bursts and intensive I/O bursts. By means of queuing network techniques, the analysis of the model leads to the definition of a generic model that allows simulation of the behaviour of several parallel architectures.

Traditionally, speedup is defined as the ratio of the elapsed time when executing a program on a single processor to the execution time when p processors are available. We extend this definition to include the number d of *I/O nodes*. Because parallel program execution can be partitioned into two distinct phases, denoted as computation burst and I/O burst, we consider the p processors devoted to the execution of the computation burst and the d I/O nodes, to the corresponding I/O burst. An I/O node can represent a disk or a SIMD processor according to the parallel machine modeled. For instance consider a hybrid architecture in which a moderate number of processors (10-20) are arranged in a MIMD way and some (even all) of them are "boosted" by massively parallel SIMD arrays, used for the

most intensive number–crunching tasks. The clusters of SIMD processors is connected to the MIMD node through a high speed I/O channel. In this scenario, several questions may arise, such as: how does the program speedup scale with the number of SIMD processors? How much does the I/O channel impact to performance of the program? Is there any advantage in using the SIMD processors or it is better to use only the MIMD processors to perform the number–crunching tasks? The approach we propose in this thesis is, to the best author's knowledge, the first attempt to address these kinds of issues.

The purpose of this thesis is to develop analytical performance models that captures the behavior of numerous applications running on a different parallel architectures. The idea of our approach is to use queueing network models. This type of model has the advantage of providing a "white–box" view of the system to study. Because the parameters of our model correspond to measurable program characteristics, we can use the model in order to estimate the execution times for different number of processors and/or I/O nodes.

Chapter 2

Parallel architectures

Computational parallelism essentially consists of using more processors (computational nodes) which cooperate through an interconnection network for the solution of a computational problem.

Any computer, sequential or parallel, executes instructions on data. A widely used way for classifying computer architectures is Flynn's taxonomy, which considers instruction stream and data stream:

1. SISD: Single Instruction, Single Data;
2. MISD: Multiple Instruction, Single Data;
3. SIMD: Single Instruction, Multiple Data;
4. MIMD: Multiple Instruction, Multiple Data.

For example, a conventional sequential computer would be classified as Single Instruction Single Data (SISD) as the processor executes at each step one instruction on one piece of data. We list the main types of parallel architectures in Figure 2.1. We do not talk about MISD architectues since they are rarely used in computing.

2.1 Single Instruction Multiple Data architectures

A parallel machine of this class consists of N identical processors (see Figure 2.2). Each processor has a local memory in which it keeps the data which it will work on. In SIMD machines, all processors simultaneously execute the same instruction, issued by the controller processor, on their local data. The processors can communicate with each other in order to perform shifts and other array operations. This approach can reduce both hardware and software complexity but is appropriate only for specialized

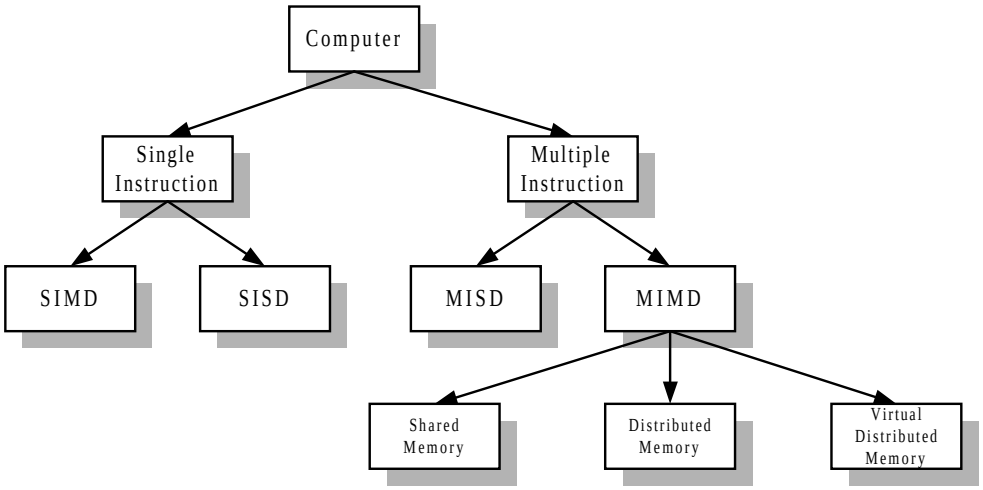

Figure 2.1: Flynn's taxonomy.

problems characterized by a high degree of regularity, for example, image processing and certain numerical simulations. If the loads on the processors are not balanced, performance is poor (because execution is synchronized at each step, with everything waiting for the slowest processor). Examples of SIMD computers are the ICL Distributed Array Processor (DAP), the Thinking Machine Corporation's CM-200, the MasPar MP and the Quadrics of QWS.

Quadrics is a high-performance machine capable of facing major computing challenges in several applied fields. Quadrics has a 3D architecture based on a cubic lattice of nodes. Each node (i.e., one processor plus its own memory) is connected to the six nearest nodes. The nodes situated at the extremity of the lattice are linked in a ring shape to those on the opposite side, obtaining, in fact, a 3D torus.

The processing element on which Quadrics is based is a proprietary floating-point processor, called MAD (multiply and add device), specifically designed to perform very efficiently the so–called "normal operation", i.e., a combined multiply–add operation ($a \times b + c$). The processor can deliver two floating–point operations per cycle. It contains a register file of 128 registers. Each MAD has 4 to 16 MB memory. The architecture of Quadrics can theoretically span from 8 to 4,096 nodes. In practice, the maximum configuration is given by 2,048 nodes, 100 Gflops (billions of floating–point operations per second) of peak power and 32 GB of memory.

As for the evolution of Quadrics technology, the Italian National Institute for Nuclear Physics (INFN) is now developing the next-generation supercomputer, named APEmille (with local addressing capability, double

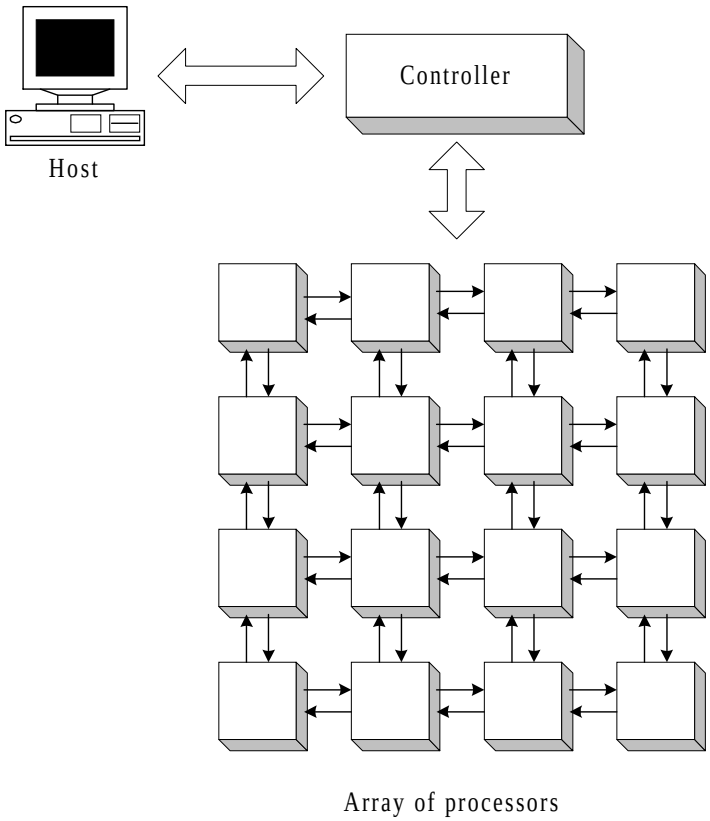

Figure 2.2: An example of a SIMD architecture.

precision, real and complex numbers, integer and logical operations, 8 operations per cycle, up to 1–2 Teraflops – trillions of floating point operations per second – of peak power).

2.2 Multiple Instruction Multiple Data architectures

This class of computers is the most general and the most powerful of the Flynn's paradigm. MIMD means that each processor can execute a separate stream of instructions on its own local data. Within this class of architectures a common method of subdividing them is on the relationship between processors and memory. This type of subdivision leads to three main types of MIMD architectures: Shared Memory, Distributed Memory and Virtual Distributed Memory.

In the shared memory architecture, shown in Figure 2.3, all processors

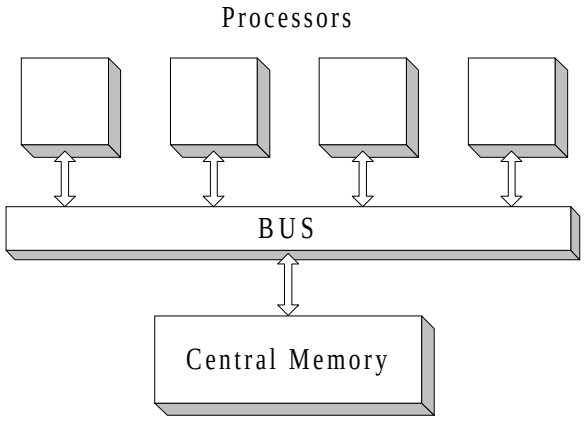

Figure 2.3: An example of a Shared Memory architecture.

access to a common memory, typically via a bus or a hierarchy of buses. The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. With this type of communication the time to access any data is the same, as all communications goes through the bus.

The advantage of this type of architecture is that it is easy to program as there are no explicit communications between processors with communications handled via the global memory. Access to this memory store can be controlled using techniques developed from multi-tasking computers, e.g., semaphores.

However, the shared memory architecture does not scale well. The main problem occurs when a number of processors attempt to access the global memory store at the same time, leading to a bottleneck. One method of avoiding this memory access conflict is subdividing the memory into multiple memory modules, each connected to the processors via a high performance switching network. However, this approach tends to simply shift the problem to the communication network.

Examples of computers with shared memory architecture are the SGI PowerChallenge, Sequent Balance and Symmetry.

The distributed memory architecture (see Figure 2.4) get around the drawbacks of the shared memory architecture by giving each processor its own memory. A processor can only access the memory which is attached directly to it. If a processor needs data which is contained in the memory of a remote processor, then it must send a message to the remote processor asking it to send it the data. Clearly access to local memory can occur much faster than access to data on a remote processor. In addition to this,

the further the physical distance to the remote processor, the longer it can take to access remote data. This non-uniform access time can be affected by the way the processors are connected. While connecting each processor to every other one is a possibility for a small number of processors, it quickly becomes impractical as the number of connections rises. One solution to the problem of connecting the processors together is to connect a processor to a small subset of its neighbors. Each of the neighbors in the communication subset would be connected to a different subset of processors, allowing for messages to be sent from one processor to another via a number of intermediate processors. There are several ways this can be done, one option would be to use switching chips which allow the user to adapt the topology of the machine to their own particular needs. Another popular possibility is to connect the processors in a hypercube arrangement. This has the advantage of not radically increased number of connections as the number of processors is increased, while offering a number of different message routing paths. Also, there are many parallel algorithms and software for hypercubes. Distributed memory machines have been built using all the methods described.

Rather than connecting processors together directly, current practice is to connect the processors to a network of routing chips. The same topology issues apply in this case but the processors no longer play any part in the message forwarding. As there can be a different number of processors and routing chips this allows greater freedom when constructing the network.

While this architecture has the drawback of requiring explicit communications, it is inherently far more scalable than the shared memory architecture which is limited by bottlenecks in accessing its global memory. Machines which have a distributed memory architecture include the Meiko Computing Surfaces.

The above classifications are in a sense idealized architectures. Often actual machines are mixtures of the different types of architectures. An example of this is the so called virtual shared memory architecture, which should be differentiated from the true shared memory machines mentioned earlier. Like the distributed memory machines, each processor has some local memory, but direct access can be made to remote memory by use of a global address space. This remote access is possible because of the incorporation of support circuitry which deals with the communication independently of the remote processor. This offers the possibility of very fast communications through the use of sophisticated hardware (though of course not as fast as local memory access) but with increasing communication overhead as the transferring distance of the messages travel increases.

An example of a Virtual Shared Memory machine is the Cray Research T3D and T3E.

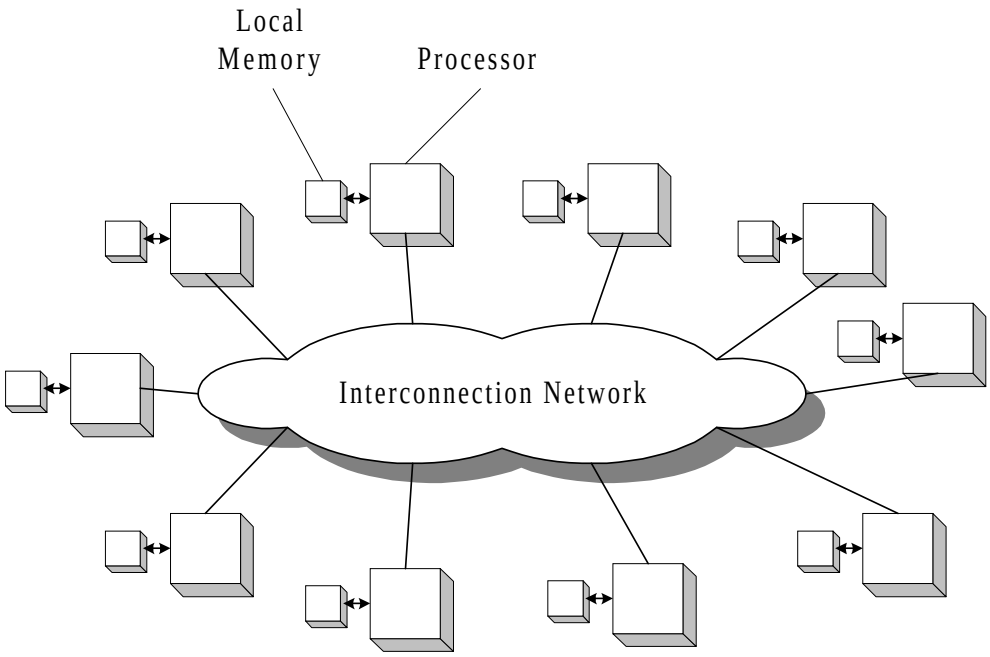

Figure 2.4: An example of a Distributed Memory architecture.

2.3 Hybrid architectures

The world of parallel computing is moving along two directions: the first depending on existing software constraints (i.e., the huge amount of legacy codes incorporating even millions of year-persons for their development, maintenance, and up-dating, over decades); the second, on ambitious goals like those called "grand challenges", where significant scientific results can be achieved only over a computing power threshold in the range of 10 to 1000 Gigaflops. In the first case, the need to be able to run "experienced" codes at increasing speeds without significantly modifying them, urges the use of shared memory based platforms that allow up to 10–20 processors to reach speedups with a satisfactory scalability. In the second case, massive parallelism is the only reasonable solution, with the consequent price to be paid in terms of revision of codes, models, algorithms, etc., in order to "map" each particular problem onto the available computational architecture.

In the frequent cases in which most of the computational weight of a code is confined in one (or a few) kernel(s), often composed of a few statements, a tradeoff can be reached between the desire of not modifying and/or customizing the code to specific platforms, and the possibility of achieving

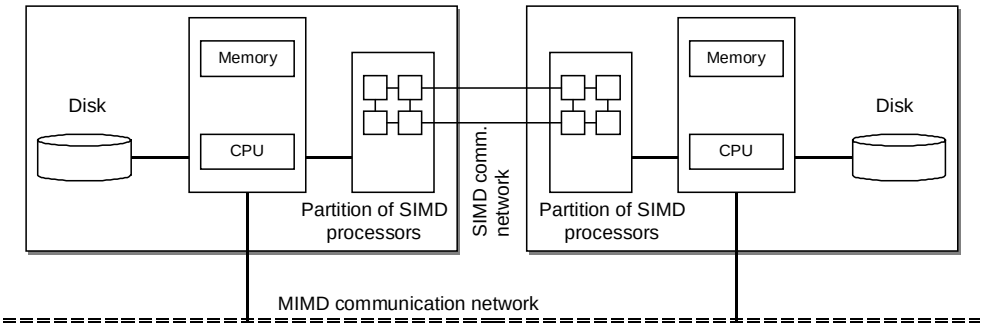

Figure 2.5: Architecture of two processing elements in a hybrid parallel computer.

more significant speedups by means of massively parallel platforms. The solution can be represented by a hybrid architecture where a moderate number of processors (10–20) are arranged in a MIMD way, while some other (even all) are "boosted" by massively parallel SIMD arrays, charged with the hugest number-crunching tasks.

Some computational problems, such as transaction processing, are dominated by integer operations and I/O. Some other, such as matrix problems or Monte Carlo simulations, are both highly parallel and synchronous. In such cases, maximum performance can be achieved by configuring the system with a moderate number of super-scalar processors, each processor with its own I/O system and a large array of simple SIMD processors. As illustrated in Figure 2.5, SIMD processors are partitioned into clusters and each cluster is connected to the MIMD node through a high speed I/O channel [10].

In order to take advantage of both the flexibility of a MIMD architecture and the scalability of a massively parallel SIMD architecture, the National Agency for New Technology, Energy, and Environment (ENEA) has recently set up a joint–venture with Finmeccanica's Quadrics Supercomputers World Ltd. (QSW) aimed to the development of the first prototype (named PQE1) of the new supercomputer. PQE1 (see Figure 2.6) is realized by complementing the Casaccia's Quadrics modules with an 8–nodes Meiko CS–2.

A new project, named PQE2000, is adopting innovative software technologies and composite MPP architectures for petaflops computing [45]. Three Italian research agencies and one industrial company – The National Research Council (CNR), ENEA, INFN, and QSW – are involved in the PQE2000 project. The definition of the hardware of PQE2000 is strongly influenced by following architectural models: coarse–grain symme-

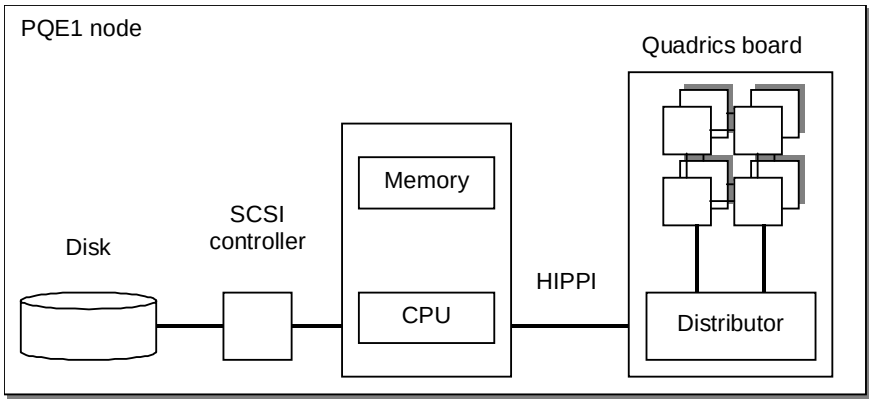

Figure 2.6: Architecture of a PQE1 node.

try multiprocessor (SMP) and uniform memory access (UMA); medium–grain nonuniform memory access (NUMA) or cache–coherent–NUMA; fine–grain SIMD; and fine–grain PIM (processors–in–memory or active memory).

2.4 Clusters of workstations

A cluster of workstations is a collection of distinct and independent high–performance workstations that are interconnected through a local area network. This configuration can be considered as a powerful bridge between computing on a single workstation and computing on supercomputers. Cluster technology takes advantages of high scalability, simple and flexible architecture, high performance–price ratio and small risk of investment.

While typically more loosely coupled than the "single box" parallel architectures described above, clusters can provide an invaluable route for producing parallel code. Furthermore, they offer the opportunity for users without resources to buy massively parallel machines to gain some of the benefits of parallelism on machines available locally. The rapidly reduction of the cost of high performance workstations/PCs makes this technology ever more available. Moreover, new concepts for the integration of individual workstations through Local Area Networks are emerging. High speed interconnection networks and optimized protocol system architectures are the most important objectives of current research in this field of study.

2.5 I/O in the parallel machines

I/O hardware parallelism essentially consists of using more disks and one or more controllers, and distribute data across the disks. Among many possible way of distributing data across multiple disks, striping is the most popular.

In a striped file system, a file is interleaved across the disks. Striping usually implies that the array of disks shares a common bus and is controlled by a single controller. Simple disk striping has performance limitations beyond five disks due to increasing overhead of managing I/O parallelism serially via a single controller [26]. In fact, modern disks have a SCSI connection and state–of–the–art controllers are capable of driving SCSI buses at no more then 20 Mbytes/sec.

A more efficient and flexible parallel disk architecture is constituted by a set of independent disks, each disk with a separate controller and connected to a distinct processor of the parallel machine. The individual devices in a parallel independent disk system could themselves be an array of disks. Usually, processors with disks are not loaded with computational tasks, but they are dedicated to the handling of I/O activities. In [37] a tracing study of all file related activity on the Intel iPSC/860, at the NASA Ames Research Center and the Thinking Machines CM-5 at the National Center for Supercomputing Applications, found that file size I/O was dominated by writes. In [5] a study of I/O characteristics of four parallel applications on a IBM SP2 using Vesta parallel file system found that I/O requests had a strong temporal and spatial locality.

The I/O represents today the limiting performance factor of large scale scientific computations that deal with large quantities of data. The impact of the I/O contention in parallel machines is becoming ever more important as the computational power of processors and the throughput of communication networks are increasing. High performance communication network and parallel file systems are needed to satisfy the data exchange requirements of current parallel scientific applications. The efficient design of such systems depends on a comprehensive understanding of the performance behavior of typical scientific applications.

Over the past two decades, advances in computer technology have led toward faster computational machines. Today high performance computers (HPC) can perform computations at gigaflop rates. Much of this computational capacity has been mainly addressed toward large–scale mathematical modeling and simulation of various physical, chemical and biological phenomena. Many computational intensive applications are among a set of scientific Grand Challenges, an annual list first initiated a decade ago by Kenneth Wilson winner of the Nobel physic prize [46].

Application	I/O storage	I/O bandwidth
4-D Earth observing	3 Tbytes 1	Tbyte/day
Particle algorithms in cosmology and astrophysics	1 Tbyte	200 Mbyte/second
Radio synthesis imaging	10 Gbytes	100 Mbyte/second
Computational quantum materials	3 Gbytes	100 Mbyte/second
High–performance aircraft simulation	4 Gbytes	100 Mbyte/second
Computational fluid and combustion dynamics	1 Tbyte	45 Mbyte/second (for visualization)

Figure 2.7: I/O requirements for Grand Challenge applications

Aside from requiring significant amount of processing time, these applications often deal with enormous quantities of data that must be exchanged among working processors and/or with I/O devices. Current high–performance applications involve 1 Gigabyte to 4 terabytes of data per run. Table 1 summarizes the I/O requirements for some Grand Challenge applications [29]. Such applications, whose bottlenecks is the I/O and not the computation, are said to be I/O bound. As a single processor and a single disk are unable to sustain the requirements of current Grand Challenge applications, a logical solution is to use more processors and more disks. In a parallel system with p processors and d disks, it is theoretically possible to achieve p times the computational rate of a single processor and to deliver d times the data of a single disk to an application. However, network and system software, as well as applications bottlenecks, greatly reduce the computational and I/O speedup.

Real parallel applications can exploit I/O for different purposes [17]:

- *Checkpoints.* Periodic checkpointing of long-run computation state is essential in order to reduce the cost of system failures. On large parallel computers, state can be large (many Gigabytes).

- *Simulation data.* Scientific and engineering simulations that compute the time evolution of physical systems periodically save system state for subsequent visualization or analysis. Some simulations can generate very large amounts of data – hundreds of Gigabytes or more in a single run.

- *Out–of–core computation.* Some programs must operate on data structures that are larger than available "core" memory. In principle, a virtual memory system can perform the necessary paging of

data to and from disk. In practice, not all parallel computers provide virtual memory.

- *Data retrieval.* Many applications involve the analysis of large amounts of data (e.g., data from weather satellites may be searched for temperature values). These data analysis applications are particularly demanding from an I/O point of view, because little computation is performed on each datum retrieved from disk.

Chapter 3

Queueing network models

In this section an abstraction technique from computer systems to queueing network models is presented. The parameters that define a model are given and the techniques used to determine exact performance estimates for the models are discussed.

3.1 Introduction

Queueing networks have been widely used to model and analyze the performance of complex systems involving service. Examples of such systems include communication systems, computer networks transaction systems, manufacturing systems and vehicular traffic. A study of such systems is necessary to evaluate their performance in terms of achievable throughputs and delays experienced. Such studies are also required to locate bottlenecks; identifying and removing the bottleneck may result in significant improvements in the performance of the system. These studies will also help in designing a system with optimum investments in resources in terms of number of servers, buffer space etc.

A queueing network is a set of interconnected queues. Customers, after receiving service at a queue, will move to another queue (or out of the network) with some probability. The service time for a customer at a queue is chosen independently for each visit to the queue, according to a service time distribution. In a queueing network, multiple classes of customers can exist simultaneously. Each class can have a different service time distribution at each of the queues in the network.

When applying queueing networks to computer system, the stations represent the various system resources (e.g. CPUs, channels, disks, etc.) while the customers represent jobs in the system.

Queueing network models are defined by servers, customer classes, and

a description of how the classes use the servers. A customer class contains one or more customers that have independent yet statistically identical behaviour. A customer class can also be referred to as a group. If there is a fixed number of customers of the class in the model at all times, it is called a closed class. Closed class customers alternate between queueing for various resources and being in a queue for idle customers. The idle time is referred to as a think time because computer system users often initiate some system function, receive a result, then think about the result before making another request for work. Idle periods can be of duration zero. A model consisting only of closed classes is a closed model. If customers are better described as arriving at some rate, satisfying their service requirements, and then leaving the system, the class is called an open class. The arrival of an open class customer is an invocation of its corresponding program. A model consisting only of open classes is an open model. If both open and closed classes are present, the model is called a mixed model.

Overall system performance can be evaluated by using parameters (service time,visit ratio, system load) of the system and its components to calculate performance measures, such as response time, throughput, and utilization. Analysis of the devices can determine their throughputs when they are operating at 100% of capacity. This is described as the saturation point of the device. The device with the lowest throughput at saturation will limit the throughput of the entire system, thus creating a bottleneck. When the system attempts to process jobs faster than the bottleneck device, system response time increases.

3.2 Solution Techniques

Many queueing networks do not have closed form analytic solutions and cannot be evaluated other than by Monte Carlo simulation. Simulations are expensive to perform, and the results which are calculated are only known to fall within certain confidence intervals. Queuing analysts found closed form solutions for increasingly complex queueing networks through the 1960s, and in 1975 the discovery of the class of separable or product form queueing networks was announced. This class of networks represents the most complex queueing networks that can be effectively evaluated analytically today.

3.2.1 Open model solution technique

Since for the open queueing network the system throughput is given as an input, solutions can be obtained simply. Let C be the number of customer classes of a given open queueing network. We denote the vector of arrival

rates for each class by $\overrightarrow{\lambda} = (\lambda_1, \lambda_2, \ldots, \lambda_C)$. Let $D_{c,k}$ denote the given service service time of the class c at center k. The response time of the queueing center k for the class c, $R_{c,k}$, is given by:

$$R_{c,k} = \frac{D_{c,k}}{1 - \sum_{j=1}^{C} U_{j,k}(\overrightarrow{\lambda})}, \tag{3.1}$$

where $U_{c,k}$ is the utilization of the center k of the class c, given by $D_{c,k}\lambda_c$. It is possible to prove [4] that (3.1) is always valid for PS and LCFS queueing discipline. For FCFS, (3.1) is valid if all classes have the same service time.

3.2.2 Closed model solution technique: the Mean Value Analysis algorithm

Queueing networks in which customers circulate between service centers have the potential to be extremely difficult to analyze because of the interdependency of the different service centers. Jackson [30] showed that open networks with all services exponentially distributed and arrivals Poisson could be analyzed as if the service centers were independent $M/M/1$ queues. Baskett et al. [4] were able to extend this analysis to a more general framework, allowing non exponential services in some instances, and also different classes of customers with their own routing behaviour. They showed the probability of the network being in a state $\overrightarrow{N} = (N_1, N_2, \ldots, N_K)$, when there are K service centers, is of the form:

$$\Pr(\overrightarrow{N}) = \frac{1}{G} \prod_{i=1}^{M} f_i(n_i). \tag{3.2}$$

If the network is open, the factor G is 1, and the service centers are essentially independent of one another, only interacting through the routing of customers to other centers after service. When the network is closed, however, the factor G, which ensures that the probabilities are normalized, introduces a dependency between the service centers. The fact that it is known that center 1, say, has 5 customers present, alters the probability of there being 5 customers at other centers. Obvious approaches to calculating G, for example by summing over all possible states, soon run into practical problems because of the number of states involved, not to say numerical difficulties such as round off.

Algorithms to numerically evaluate these networks have been the subject of much interest. Buzen [6] developed the first algorithm, known as the convolution algorithm. This algorithm finds the normalization constant

G for a network of K centers and N customers using a simple recurrence relating $G(M,N)$ to $G(M-1,N)$ and $G(M,N-1)$. Other performance metrics, such as mean queue lengths, center utilizations, etc. are found using G. Although very efficient, the convolution algorithm is not very intuitive, and its computations can be affected by overflow or underflow for large networks. Reiser and Lavenberg [40] developed a new algorithm, Mean Value Analysis (MVA), that uses only meaningful metrics of network performance in its calculation. Essentially, the performance of the network when N customers are present is evaluated using the performance of the network when there are $N-1$ customers. Metrics, such as utilization and mean queue length, are produced as a side effect. The normalization constant is not calculated. MVA is of similar complexity to convolution.

Mean value analysis operates by relating the performance of the network when n customers are present to the performance when $n-1$ are present. Since the performance when there are 0 customers is known trivially, calculation proceed using increasing populations, from 0 to N. When the population is N, and there are K stations, the complexity of the algorithm is $O(KN)$. If there are C classes of customer, then the performance when the population vector is $\overrightarrow{n} = (N_1, N_2, \ldots, N_C)$ is calculated using the performance at populations $\overrightarrow{n} = (N_1 - 1, N_2, \ldots, N_C), (N_1, N_2 - 1, \ldots, N_C), \ldots, (N_1, N_2, \ldots, N_C - 1)$. Given a final population for which we wish to calculate the performance, the calculations needed give a precedence relationship between the different populations. The population 0 precedes all other populations, and the other populations must be calculated. The order of calculation is not totally determined, since the precedence relationship is only a partial ordering.

In order to describe more formally the MVA algorithm we use the following notations:

- C, number of customer classes;
- K, number of service centers;
- N_c, population of the class c;
- $Q_k(\overrightarrow{n})$, queueing length of the center k for the population $\overrightarrow{n} = (n_1, n_2, \ldots, n_C)$;
- $D_{c,k}$, service demand of the class c at center k;
- $U_{c,k}$, utilization of the class c at center k;
- $R_{c,k}$, response time of the class c at center k;
- X_c, throughput of the class c;

for k:=1 **do** $Q_k(\overrightarrow{0})$:=0;

for n:=1 **do** $\sum_{c=1}^{C} N_c$

for each $\overrightarrow{n} \equiv (n_1, n_2, \ldots, n_C,)$ **such that** $\sum_{c=1}^{C} n_c = n$ **do**

begin

for c:=1 **to** C **do**

for k:=1 **to** K **do** $R_{c,k} := D_{c,k}\,[1 + Q_k(m_c(\overrightarrow{n}))]$;

for c:=1 **to** C **do** $X_c := \dfrac{n_c}{Z_c + \sum_{k=1}^{K} R_{c,k}}$;

for k:=1 **to** K **do** $Q_k(\overrightarrow{n}) := \sum_{c=1}^{C} X_c R_{c,k}$;

end;

Figure 3.1: MVA solution algorithm.

- Z_c, think time of the terminal of class c (the sum of all class c delays);
- $m_c(\overrightarrow{n})$, population $\overrightarrow{n}$ with one class c customer removed, i.e., $(n_1, n_2, \ldots, n_c - 1, \ldots, n_C)$.

The exact MVA solution algorithm [33], in psuedo-code, is shown in Figure 3.1. When this algorithm terminates, the values of $R_{c,k}$, X_c and Q_k are available.

In chapter 6 we will show a parallel version of algorithm MVA, and will study its performance.

Chapter 4

Integrate models

In parallel architectures, several processors work simultaneously and cooperate to the execution of a single task. In this way, the computing performance can be significantly increased compared to a sequential single–processor architecture. Performance predictions for parallel programs on multiprocessor systems are therefore of crucial importance for both software and hardware designers.

In this chapter we formulate a general model of program behavior that captures the computation and I/O characteristics of a parallel application, or a class of parallel applications[1]. We then derive a simple mathematical analysis of the model, and define performance metrics based on this analysis. The multiprocessor system under consideration has a generic structure, as the one illustrated in Figure 4.1. We assume a fixed number of (homogeneous) processors and a fixed number of (homogeneous) I/O nodes. Computational nodes and I/O nodes are connected to each other via a communication network. Through this model we can study the influence on the speedup of communication aspects (synchronizations, link contentions, scaling factors) and of I/O issues (synchronizations, data managing before I/O, time spent sending the data to the I/O nodes).

We consider two distinct hardware configurations of I/O systems:

- BUS: I/O nodes are connected via a single bus to the system (Figure 4.2a);
- CLU: cluster of processors that shares a common I/O node (Figure 4.2b).

The BUS case is a centralized architecture in which a single I/O node, or a pool of I/O nodes, are connected via a "single" path to the processors.

[1] Throughout the discussion we consider a monoprogrammed multiprocessor system, i.e., there is always only one program is execution.

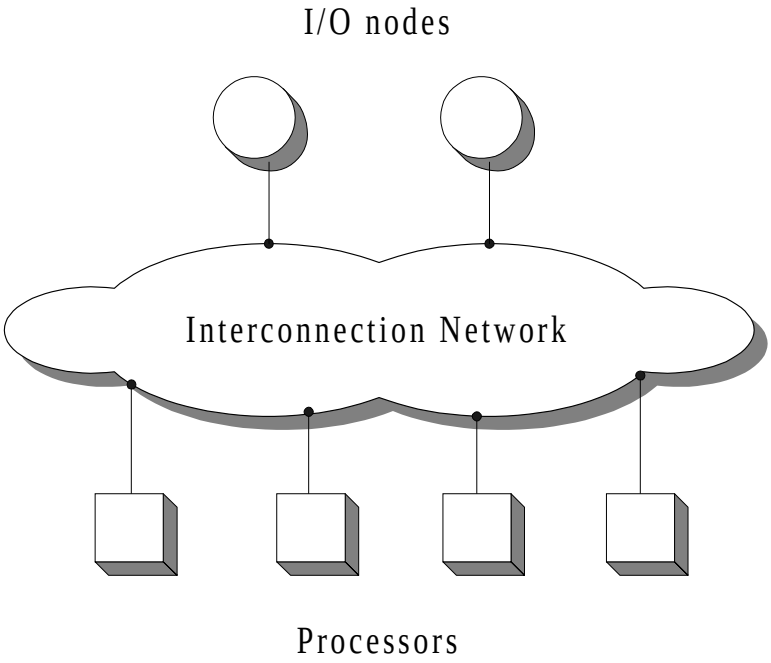

Figure 4.1: General purpose parallel machine.

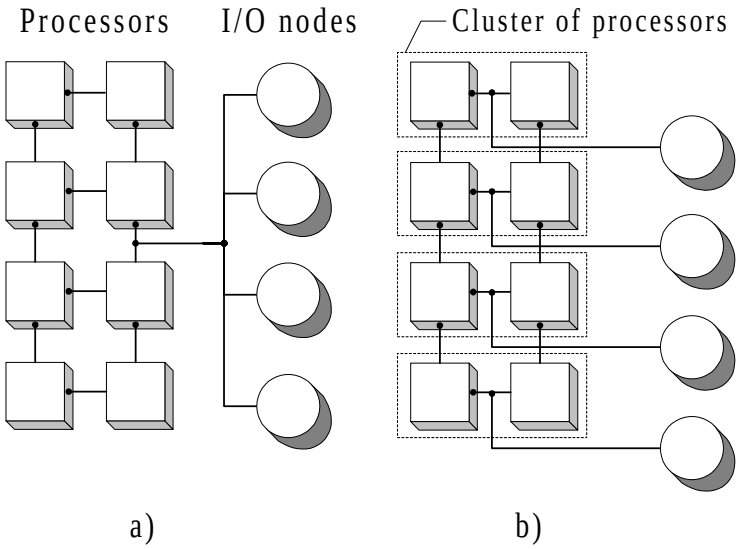

Figure 4.2: I/O configurations considered: a) BUS, b) CLU.

Processors accessing the I/O nodes must send/receive data to/from the gateway processor that manages the I/O nodes. Several components could become the bottleneck of the I/O. By increasing the number of I/O nodes we expect that the performance of the I/O improve, however any further increase of the number of I/O nodes besides a certain limit will saturate the bus or the gateway processor and do not produce any increase in the throughput.

In the CLU case, a pool of processors shares an I/O node. When the number of I/O nodes is equal to the number of processors we have one I/O node for one processor.

4.1 Parallel program behavior

Studies on I/O characterization of scientific applications [37, 5, 35, 39] showed that I/O behavior is rather regular and cyclic along time. The I/O properties of many parallel programs result in execution behavior that can be naturally partitioned into disjoint intervals, each of which consists of a single I/O burst followed by a single computation burst (see Figure 4.3):

- *Computation burst*: is partitioned into disjoint sub–intervals each one consisting of a single burst of CPU activity (pure calculation operations) followed by a single burst of communication (inter–processors data exchange);

- *I/O burst*, I/O read/write operations.

We use the term *phase* to refer to each such interval composed of an I/O burst followed by a computation burst.

4.2 Speedup models

In this section we develop a performance model for a program executed on a system composed of a single processor and I/O node. We use this model as a *reference model*. Let $p_{\mathrm{I/O}}$ be the probability of an I/O burst at the end of a CPU burst, we have that the global computation burst is composed by $1/p_{\mathrm{I/O}}$ CPU bursts. Where the average time of the computation burst is given by S_{p}. The reason for this representation will be clarified when we will introduce the communication overhead. Figure 4.4 depicts the closed queueing network corresponding to the reference model. A single job, representing the program in execution, circulates in the queueing network. The average response time of the circulating job can be expressed

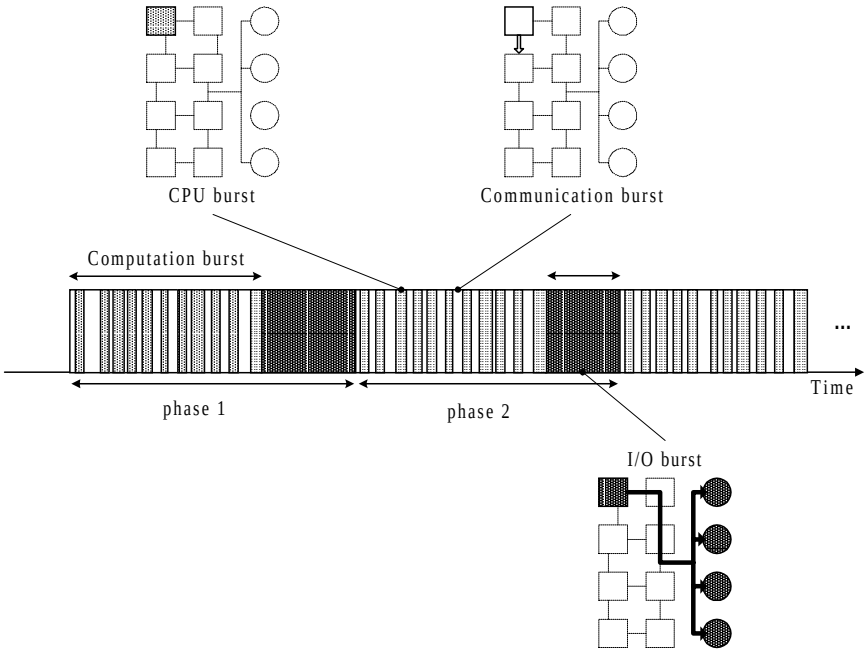

Figure 4.3: An example of program behavior.

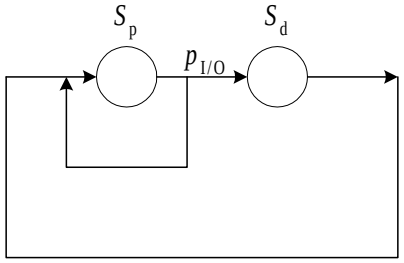

Figure 4.4: Queueing network model of a program executed on a single processor and I/O node.

as[2]:

$$T_0 = N\left(\frac{S_{\mathrm{p}}}{p_{\mathrm{I}/O}} + S_{\mathrm{d}}\right), \tag{4.1}$$

where N represents the average number of phases of the program. Note that the term $S_{\mathrm{p}}/p_{\mathrm{I}/O} + S_{\mathrm{d}}$ is the response time of the queuing network, i.e., the average execution time of one phase of the program. Throughout the thesis we will use T_0 as a reference time, in order to evaluate the speedup behavior of different parallel programs and architectures.

Suppose we have a parallel program executed on a parallel machine with p processors and d I/O nodes. We decompose its average execution time $T(p,d)$ into two terms:

$$T(p,d) = N(T_{comp}(p,d) + T_{\mathrm{I}/O}(p,d)), \tag{4.2}$$

where $T_{comp}(p,d)$ and $T_{\mathrm{I}/O}(p,d)$ represent the average time of the computation burst and I/O burst, respectively. The normalized values of these two times $\tau_{comp}(p,d)$ and $\tau_{\mathrm{I}/O}(p,d)$ are defined as follows:

$$\tau_{comp}(p,d) = \frac{T_{comp}(p,d)}{\dfrac{S_{\mathrm{p}}}{p_{\mathrm{I}/O}} + S_{\mathrm{d}}}$$

and

$$\tau_{\mathrm{I}/O}(p,d) = \frac{T_{\mathrm{I}/O}(p,d)}{\dfrac{S_{\mathrm{p}}}{p_{\mathrm{I}/O}} + S_{\mathrm{d}}}.$$

Throughout the discussion, we indicate the normalized times by small Greek letters. From (4.2) we can formulate the speedup[3] law $s(p,d)$ of parallel programs that represents a generalization of Amdahl's law [42] extended to a system with multiple I/O nodes:

$$s(p,d) = \frac{T_0}{T(p,d)} = \frac{\dfrac{S_{\mathrm{p}}}{p_{\mathrm{I}/O}} + S_{\mathrm{d}}}{T_{comp}(p,d) + T_{\mathrm{I}/O}(p,d)} = \frac{1}{\tau_{comp}(p,d) + \tau_{\mathrm{I}/O}(p,d)} \tag{4.3}$$

In order to normalize the times which characterize our queueing network models, we use the time $S_p/p_{\mathrm{I}/O} + S_{\mathrm{d}}$ (i.e., the time required by one phase of the program executed on single processor and I/O node). Let τ_{CPU}

[2]With *response time* we will refer to the average time required by a job to perform a cycle through the network.

[3]Our definition of speedup should be called *relative speedup*. However, throughout this thesis we will refer to it dropping the word "relative".

and τ_{comm} be the respective (normalized) global durations of the CPU and communication bursts ($\tau_{comp} = \tau_{CPU} + \tau_{comm}$). The speedup s can be expressed as:

$$s(p,d) = \frac{1}{\tau_{CPU}(p,d) + \tau_{comm}(p,d) + \tau_{I/O}(p,d)}. \tag{4.4}$$

The advantage of using normalized times is that we can evaluate the speedup of a program by simply calculating the inverse of its execution time. Indeed, in this way we consider the average execution time of the same program, on a single processor and I/O node, equal to 1.

Figure 4.5 depicts the generic queueing network used for modeling the parallel program behavior described in the previous section. The blocks represent the "sub–models" corresponding to the bursts of the program phases. A single job, representing the running program, circulates in the queueing network. According to the type of I/O used by parallel programs, we can have:

- SIO (Synchronous I/O): all processors allocated to the program perform I/O bursts always at the same time (e.g., checkpoint and simulation data).
- AIO (Asynchronous I/O): the processors allocated to the program perform I/O bursts at different instants of time (e.g., out–of–core).

We model the SIO cases using a pair of *fork/join* stations. A single job represents the I/O activity. When this job is collected by the fork station it is replaced with a certain number of jobs which correspond to the computation burst activities. On the contrary, when all jobs, representing the computation burst activities, are collected by the join station, they are replaced by a single job that corresponds to the I/O burst activity.

In the case of AIO models the jobs representing the computation burst activities perform I/O bursts independently. Hence, as shown in Figure 4.5, the fork/join pair are removed from the computation burst block of the AIO model.

4.3 Computation burst model

4.3.1 Introduction

Among different interconnection networks, two extreme situations can be identified: a fully interconnected system and a single bus system. In the former case, no contention arises in any communication operation: messages exchanged between pairs of processors experience a simple delay regardless

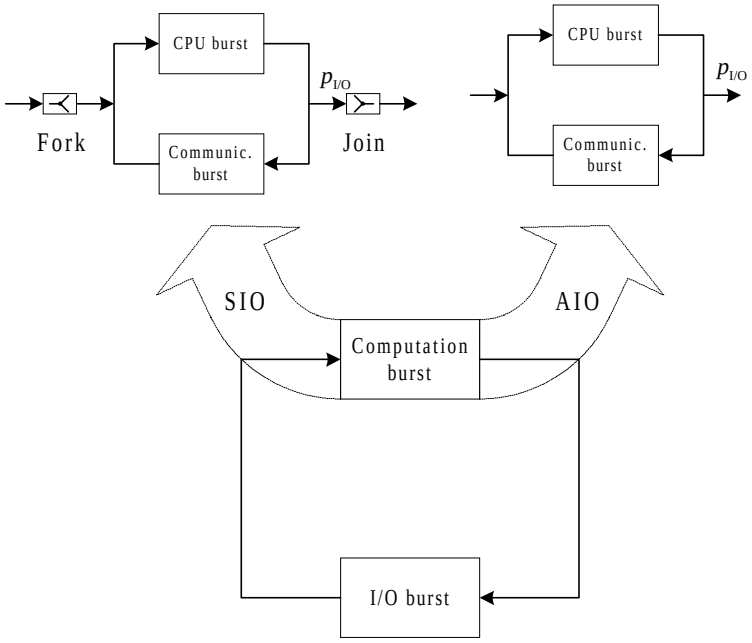

Figure 4.5: Structure of the queueing network used for modeling parallel programs.

of the network usage. In the latter case, when two or more messages are exchanged simultaneously, there is always contention for the use of the shared bus. Different interconnection networks, as for example meshes or trees, represent tradeoffs between the communication performance and the scalability of the machine. Moreover, it is clear that the impact of the communication depends also on the software. For instance, a pipeline implemented on a parallel machine can be realized in such a way that the communication does not suffer from contention for the communication network. However, little can be done in order to avoid that the processors waste time during the communication while they wait for their receiver/sender partners. Even when the balance of the work among the processors is perfect, there can be a little difference of computation time (e.g., due to the processor caches) that puts the processors out of phase.

A common programming paradigm in scientific applications, when executed on distributed memory architectures, involves the decomposition of the problem domain. Data decomposition is often used as a method for obtaining some degree of parallelism that is usually easy to manage and typically matches the problem structure closely. This decomposition is translated into a data–domain mapping over the set of computational nodes. Each node applies a sequence of similar operations to all or most el-

ements of the local portion of the distributed data structure. Such parallel applications generally use several N–dimensional arrays that are distributed in a block fashion among processors. Main advantage of this approach is the simplicity (thus, the rapidity) of the code development; main drawback is the concurrency of communications and I/O operations which causes the interconnection network and/or the I/O system to become the bottleneck of the system.

In the following sections, we formulate a general model of a parallel program that attempts to address these issues. In sections 4.3.2 and 4.3.3 we define the model parameters which represent the communication times. In sections 4.3.4, 4.3.6 and 4.3.5 we present the model parameters that take into account the communication scaling factor, the network contention and the synchronization during the processor communications. Finally, section 4.3.7 presents the queueing network that models the computation burst activity.

4.3.2 Communication transfer time

The main parameter of the communication is the transfer time σ_{r}. It represents the time that the processor, of the target parallel machine, spends to transfer a certain amount of data point to point to another processor of the same machine. Often, this time is referred in literature as the ratio of the number N of bytes of the data to transfer by the bandwidth T_b of the communication network, expressed in byte/sec. Since σ_{r} is a normalized time, it corresponds to the percentage of total communication time with respect to the program execution time on a single processor and I/O node.

4.3.3 Communication startup time

We must also consider a parameter that models the communication startup time, which is practically present in all networks. We use the parameter $\sigma_{\mathrm{r}0}$ (a real number ≥ 0) in order to represent the normalized communication startup time.

4.3.4 Dimensions of data space

Generally, the amount of inter-processors communication depends on the hyper–perimeter of the distributed data structure (while the computational load usually depends on the hyper–volume). For instance, in low–level image processing applications, the working data structure is a two–dimensional matrix of pixels, usually partitioned into square blocks, or windows, among the processors. The size of the messages exchanged among processors is proportional to the length of the internal border of adjacent

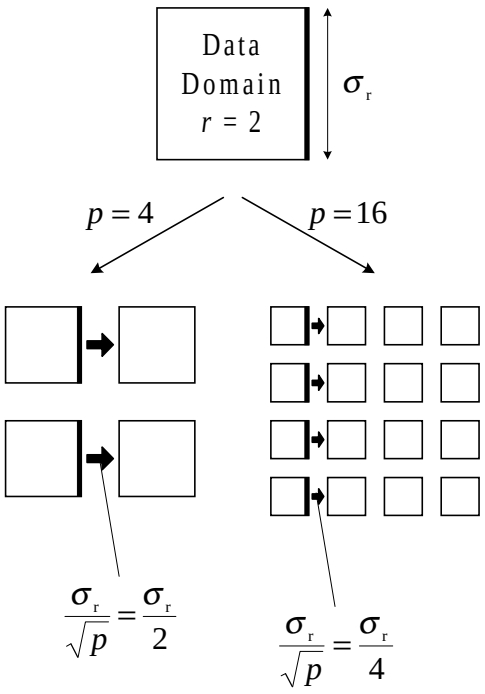

Figure 4.6: Example of communication scaling factor for a bidimensional data domain.

windows. Thus, the size of message is reduced by a factor of $\sqrt{p}$ (see Figure 4.6).

We can generalize this idea for a generic number of dimensions r of the program data space, introducing a scaling function $g(p)$, so that the average communication time per processor is given by $\sigma_r g(p)$. The choice of the scaling function $g(p)$ depends on how the data domain is partitioned among the working processors. Through the scaling function $g(p)$ we are able to capture a broad class of parallel applications. For instance, when $g(p) = 1$, we suppose that the communication factor does not scale with the number of processors (it is the case of a data domain partitioned along one dimension). Since the ratio between the hyper–volume and the hyper–perimeter of the hyper–cube, representing the program data, is $p^{\frac{r-1}{r}}$, we have:

$$g(p) = 1/p^{\frac{r-1}{r}}. \tag{4.5}$$

As particular cases we often consider two extreme situations, $r = 1$ and $r \to \infty$, which correspond to $g(p) = 1$ and $g(p) = 1/p$, respectively.

4.3.5 Synchronization level

In this section we define a parameter that captures another important behavior: the synchronization among the processors during the communication. The number of processors c, an integer $1 \leq c \leq p$, involved in the synchronous communication is referred as *Synchronization Level*. When $c = 1$ we assume that the processors perform asynchronous communications. When $c > 1$, it represents the number of processors involved in synchronous collective communications. For instance, if $p = 8$ and $c = 2$, then we have $p/c = 4$ groups, composed by $c = 2$ processors. The pair of processors, belonging to the same group, communicate to each other by means of synchronous send/receive operations.

4.3.6 Network contention level

In the previous sections we have not considered the communication contentions. Indeed, the term $g(p)\sigma_{\mathrm{r}}$ represents only the time spent transferring data. Different types of interconnection networks lead to different ways of modeling. For instance, a bus interconnection can be modeled as a queueing server, since it is capable of handling one message at a time, while a fully interconnected network can be modeled as a delay center since the messages never queue for a link. More realistic network topologies (e.g. 2D meshes, hypercubes, toruses) are more complex to model (they require the exact knowledge of the network routing mechanism).

We define the *Communication Contention Level* w, a real number $0 \leq w \leq 1$, that allows us to switch from the bus interconnection network architecture ($w = 1$) to the fully connected one ($w = 0$). The intermediate values represent different network architectures such as meshes, trees, etc., in which the communication among the processors has delays due to the occupation of the connection network. The value of w depends on the communication hardware and on the program algorithm. For instance if the processors of a program that is executed on a machine with a mesh network topology communicate always with their neighbors, the network behaves like a delay, i.e., $w = 0$.

4.3.7 The model

We now exploit the parameters above defined in order to develop a queueing network model that represents the time spent by a program during a computation burst. As mentioned before, we refer to this model as computation burst sub–network and we use it inside the various models presented in the following sections. Figure 4.7 shows this sub–network only for the case of asynchronous I/O (AIO), while for the case of synchronous I/O

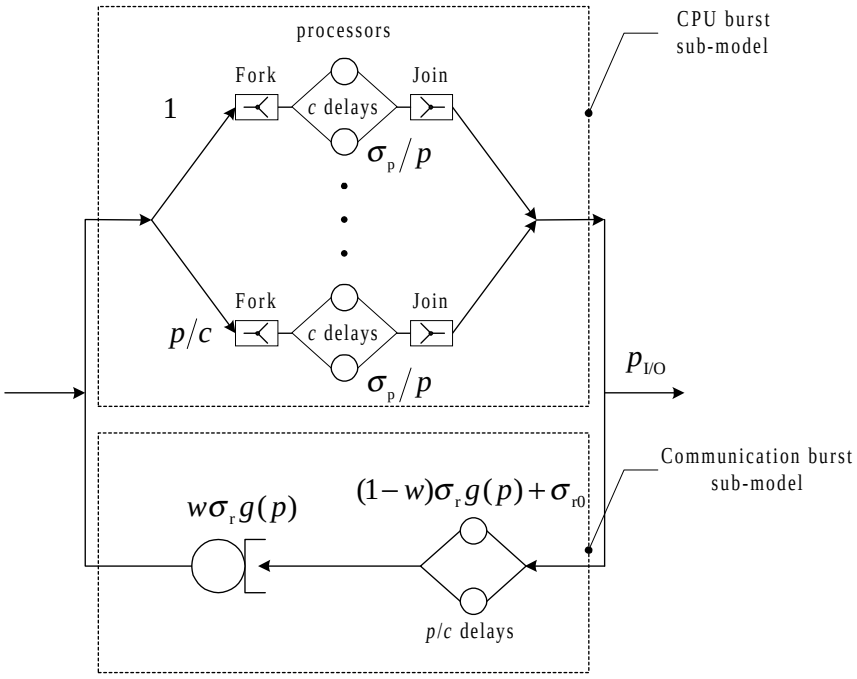

Figure 4.7: Sub–network that simulates the computation burst.

(SIO) it is sufficient to include the fork/join pair in the model, as depicted in Figure 4.5.

A job entering in the sub–network represents a group of c processors which communicates among themselves by means of collective communication operations. Each job proceeds first to a set of delays that models the processors, then proceeds to the two service centers representing the communication network (the communication sub–model). On average, a job performs a certain number of cycles through the sub–network, given by the probability $p_{I/O}$, and then exits the sub–network.

As shown in Figure 4.7, the processors elaboration is represented by p/c delays and as many pairs of fork/join stations simulating the synchronizations[4].

A job arriving in the fork station is split into c new jobs each of them representing a single processor computation. At this stage, the c jobs are collected by the same number of delay stations. Each of them will perform a delay (given by σ_p/p) that represents pure calculation. Finally, the join station collects the jobs arriving from c delay stations and, when all the

[4]When we have more than one jobs, the symbol of the delay is replaced by a group of delays equal to the number of jobs that the delay station can serve. Throughout the thesis we simply refer to this group of delays as delay.

jobs are received, it replaces them by a new single job representing the group of synchronized processors.

The conditions $p = c$ and $c = 1$ correspond to special cases. The former represents the situation in which all processors are synchronized before each communication, i.e, there is only one job in the sub–network. The latter corresponds to the case in which all processors perform asynchronous communications.

The time elapsed between the arrive of the job in the fork station until that job departs the join station, is a random variable defined as $X = \max\{X_1, \ldots, X_c\}$. Where $X_1, \ldots, X_c$ are random variables exponentially distributed with the same mean $\frac{\sigma_{\mathrm{p}}}{p}$, representing the c processor delays. X is then a hypoexponential distributed random variable with mean given by [44]:

$$E[X] = \frac{\sigma_{\mathrm{p}}}{p} \sum_{i=1}^{c} \frac{1}{i}.$$

Therefore, the response time of the fork/join pair, is given by $h(c)\frac{\sigma_{\mathrm{p}}}{p}$, where

$$h(c) = \sum_{i=1}^{c} \frac{1}{i}. \tag{4.6}$$

The communication burst sub–model is composed by two service centers: a delay and a queuing center. We now analyze how the parameters, defined in the previous sections, are mapped into this sub–model (see Figure 4.7). The startup time σ_{r0} becomes a simple delay of the communication burst. The communication time σ_{r} is split into two terms: $(1 - w)\sigma_{\mathrm{r}}$ and $w\sigma_{\mathrm{r}}$. The former corresponds to the time of the delay station, the latter to the service time of a queueing station. In this way we can switch, as mentioned in section 4.3.6, from the case without contention ($w = 0$), i.e., in which the connection network behaves as a pure delay station, to the case with maximum contention ($w = 1$), i.e., in which the connection network behaves as a pure queueing station. The values of w such that $0 < w < 1$ represent intermediate levels of communication network contention.

In principle, we should consider the fact that σ_{r0} and σ_{r} are functions of c. This dependence, however, is not known a priori and is influenced by factors strongly dependent on the language used, on the type of collective operation (broadcast, gather, scatter, etc.) and on the hardware of the parallel machine. For these reasons we introduce the simplification $\sigma_{\mathrm{r0}} = const$ and $\sigma_{\mathrm{r}} = const$.

4.4 I/O models

It is difficult to provide a general discussion of parallel I/O because different parallel computers have radically different I/O architectures and hence parallel I/O mechanisms.

Our I/O model comprises a simple delay, equal to $\sigma_{\rm n}$, followed by a queueing station with service time equal to $\sigma_{\rm d}$ multiplied by a scaling function.

For BUS models the delay station can simulate, for instance, the time spent by the system transferring data from the processors to the I/O nodes.

For CLU models we can think the delay as the time (if any) spent by the parallel program to redistribute the data to transfer before the I/O operation. For example, if distributions on disk and in memory differ, then a large number of reads or writes may be required in order to achieve data transfer. This problem is analogous to what happens when transferring data structures between two parallel program components that require different distributions. In this situation at least two approaches are possible: we can modify one or both components to use different distributions, or we can explicitly redistribute data before or while transferring it. Because I/O requests tend to be more expensive than interprocessor communications, it is often better to perform an explicit redistribution of data in memory so as to minimize the number of I/O requests. This leads to a two–phase access strategy, in which the data distributions used on disk and in memory are decoupled.

The service time of the queueing station represents the time spent by the I/O node managing the data. If the I/O nodes correspond to disks it represents the time spent to read/write the data from/to the disks. While, if the I/O nodes correspond to SIMD processors, it represents the time spent to execute the SIMD procedure. We will always assume that this service time scales as $1/d$, since the data are "striped" across the I/O nodes.

4.5 Model parameter summary

The parameters used throughout the thesis are summarized in Table 4.8.

The parameters which refer to times in the computation burst (i.e., $\sigma_{\rm p}$, $\sigma_{\rm r0}$ and $\sigma_{\rm r}$) are normalized with respect to the time $S_{\rm p} + p_{{\rm I}/O} S_{\rm d}$. The parameters $\sigma_{\rm d}$ and $\sigma_{\rm n}$ are instead normalized with respect to $S_p/p_{{\rm I}/O} + S_{\rm d}$. As consequence, all the previous parameters represent fractions of time with respect to the time required by the program executed on a single processor and I/O node.

Parameter	Meaning	Type
p	number of processors	Integer $[1, +\infty[$
d	number of I/O nodes	Integer $[1, +\infty[$
σ_{p}	computation time	Real $]0, 1]$
σ_{r0}	communication startup time	Real $[0, +\infty[$
σ_{r}	communication time	Real $[0, +\infty[$
σ_{n}	time spent transferring I/O data or spent redistribute I/O data	Real $[0, +\infty[$
σ_{d}	I/O node time	Real $= 1 - \sigma_{\mathrm{p}}$
r	number of dimensions of the program data space	Integer $[1, +\infty]$
w	Communication Contention Level (0 = delay, 1 = queue)	Real $[0, 1]$
c	Synchronization Level (number of processor	Integer $[1 \ldots p]$

Figure 4.8: Models parameters summary

Chapter 5

Model analysis

In the preceding chapter we have shown the models corresponding to various software/hardware aspects of the parallel applications. In this chapter we will describe how to use these models and we will develop and illustrate the algorithm required to evaluate the models.

Section 5.1 presents two optimistic situations in which we neglect the communication and $\sigma_{\rm n}$. In sections 5.2, 5.3, 5.4 and 5.5 we describe four models obtained by combining the two possible hardware configurations (i.e, BUS and CLU) with the two types of I/O (i.e., SIO and AIO). In section 5.6 we present two simple case studies.

5.1 The optimistic cases

We consider optimistic situations in which we neglect the effects of communication and of time $\sigma_{\rm n}$. This analysis leads to two models: the first one for AIO programs and the second one for SIO programs. These models provide simple upper bounds of the speedup that can be obtained from real parallel programs.

The model for optimistic AIO programs (Figure 5.1) consists of two delay stations. A delay $\sigma_{\rm p}/p$ represents the processor computations and a delay $\sigma_{\rm d}/(dp)$ represents the I/O nodes time. The latter delay scales with $1/p$ since we assume that the time $\sigma_{\rm d}$, corresponding to the whole data set, is distributed among the processors. From this model we can formulate a simple relation that expresses the normalized average execution time for one program phase:

$$\tau_{AIO}(p,d) = \frac{\sigma_{\rm p}}{p} + \frac{\sigma_{\rm d}}{dp}. \tag{5.1}$$

From (5.1) we can obtain the speedup as a function of number p of proces-

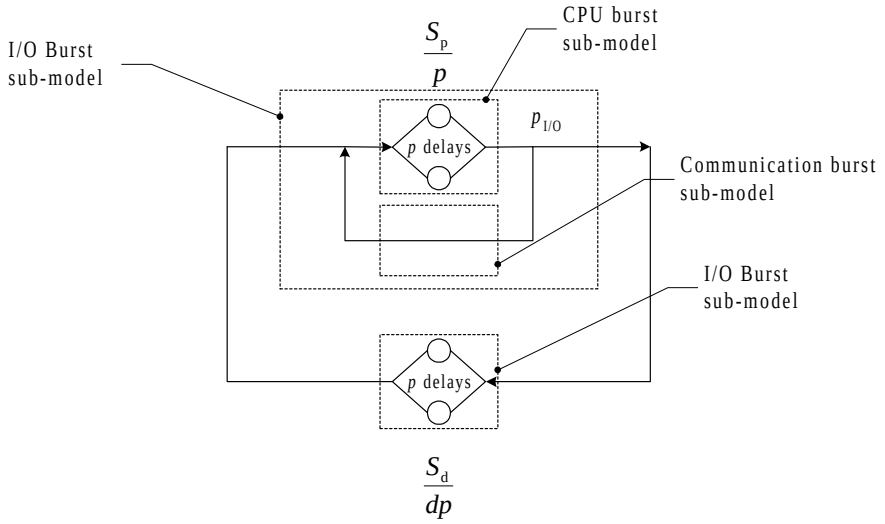

Figure 5.1: Queuing network model of the AIO optimistic case.

sors and the number d of I/O nodes:

$$s_{AIO}(p,d) = \frac{1}{\frac{\sigma_p}{p} + \frac{\sigma_d}{dp}} \tag{5.2}$$

As discussed in the preceding chapter, in SIO models we must introduce the fork/join pair, in order to take into account the effect of I/O synchronizations (see Figure 5.2). The effect of the fork/join on the speedup is achieved by means of factor $h(c)$ defined in (4.6). Therefore, the speedup can be then expressed as:

$$s_{SIO}(p,d) = \frac{1}{h(p)\frac{\sigma_p}{p} + \frac{\sigma_d}{d}} \tag{5.3}$$

Figures 5.3 and 5.4 show the speedups obtained by using (5.2) and (5.3), respectively. They should be interpreted as upper bounds of the corresponding speedup surfaces which will be presented throughout this chapter. However, it is already possible to see the huge difference between the speedup of AIO and the speedup of SIO. Nevertheless, as described in section 4.2, the purpose of these two types of I/O is different.

Note that from our definitions it is easy to see that $\sigma_p + \sigma_d = 1$. Therefore σ_p and σ_d represent the respective percentage of processor computation time and I/O time, with respect to the total execution time on a sequential architecture.

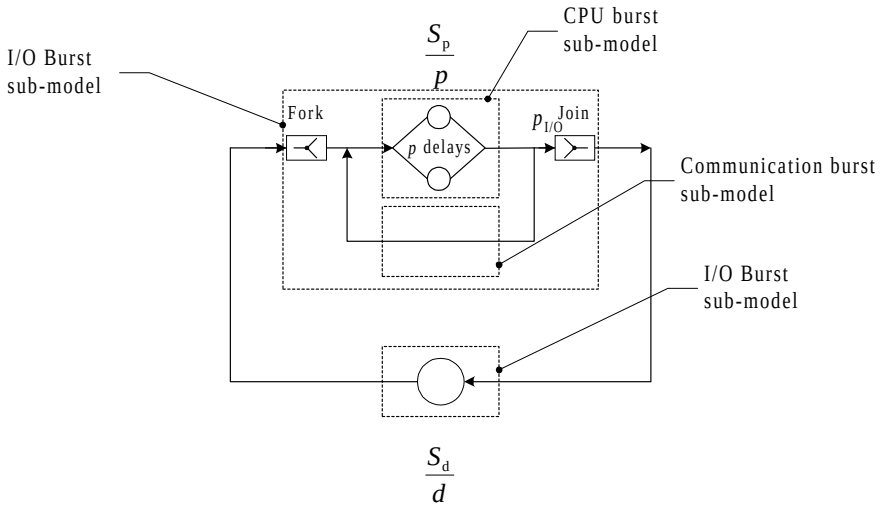

Figure 5.2: Queuing network model of the SIO optimistic case.

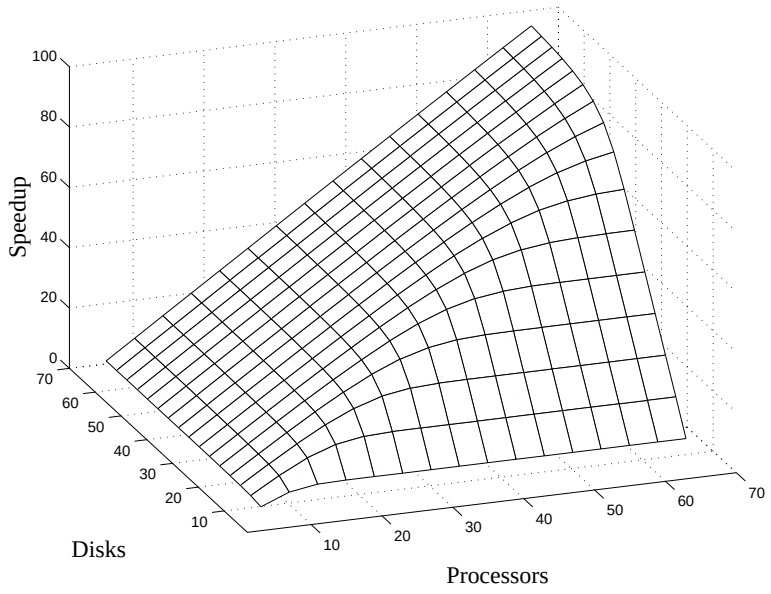

Figure 5.3: Optimistic AIO speedup surface. $\sigma_{\mathrm{p}} = 0.625$, $\sigma_{\mathrm{r0}} = \sigma_{\mathrm{r}} = 0 = \sigma_{\mathrm{n}} = 0$.

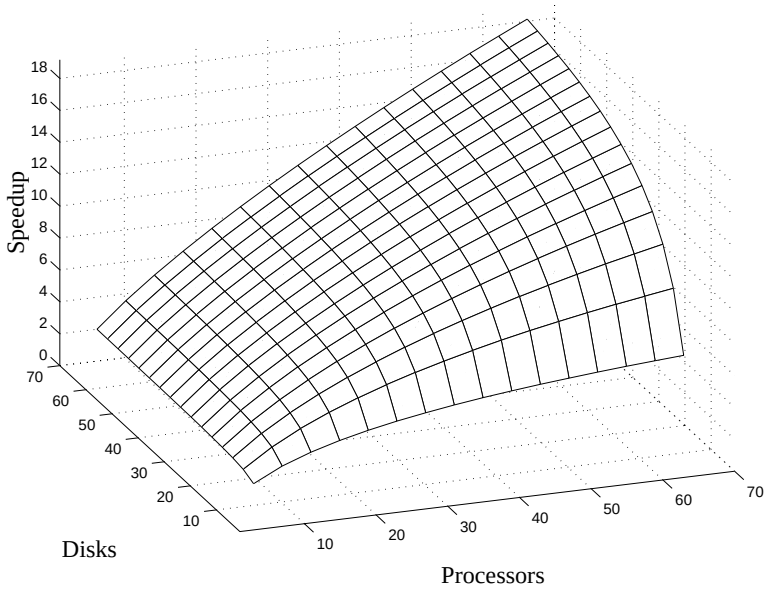

Figure 5.4: Optimistic SIO speedup surface. $\sigma_{\mathrm{p}} = 0.625$, $\sigma_{\mathrm{r0}} = \sigma_{\mathrm{r}} = 0 = \sigma_{\mathrm{n}} = 0$

5.2 BUS–SIO Model

BUS–SIO model simulates parallel programs which performs synchronous I/O, executed on architectures with centralized I/O strategy. An example of this scenario is an architecture with redundant disk arrays (RAID disks) or an hybrid MIMD+SIMD machine. Figure 5.5 depicts the BUS–SIO model. In order to determine the speedup of BUS–SIO model, we first evaluate the time τ_{comp} (i.e., the time spent by the program during the computation burst), by estimating the computation burst sub–network response time.

Let $D(p)$ be the service time of communication queueing station given by:

$$D(p) = w\sigma_{\mathrm{r}} g(p), \tag{5.4}$$

and let $Z(p)$ be the sum of all delays belonging to the computation sub–network:

$$Z(p) = h(c)\frac{\sigma_{\mathrm{p}}}{p} + \sigma_{\mathrm{r0}} + (1-w)\sigma_{\mathrm{r}} g(p), \tag{5.5}$$

we have [20] (see Appendix B):

$$\tau_{comp}(p) = \sum_{i=1}^{p/c} \frac{1}{i} R(i, Z(p), D(p)) \qquad p \geq 2, \tag{5.6}$$

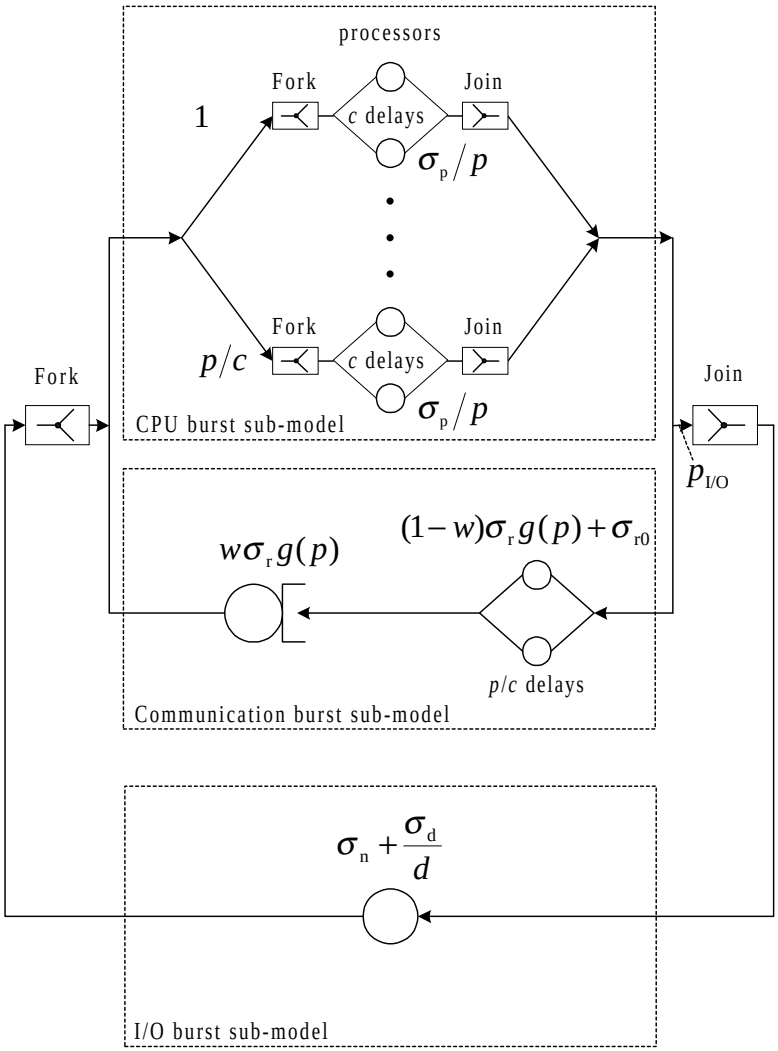

Figure 5.5: BUS–SIO queueing network model.

where $R(i,z,x)$ is the response time of a closed queueing network with population i composed of a delay z and a queueing station with service time x. The response time $R(i,z,x)$ is given by (see Appendix A):

$$R(i,z,x) = z + d\frac{\displaystyle\sum_{(e_1,e_2)\in L_{i-1}^2} \frac{e_1+1}{e_2!}x^{e_1}z^{e_2}}{\displaystyle\sum_{(e_1,e_2)\in L_{i-1}^2} \frac{1}{e_2!}x^{e_1}z^{e_2}} \tag{5.7}$$

where L_i^2 is the set of pairs (e_1,e_2) of non–negative integers such that $e_1+e_2=i$.

The time of the I/O burst is given by the following equation:

$$\tau_{\mathrm{I}/O}(d) = \sigma_{\mathrm{n}} + \frac{\sigma_{\mathrm{d}}}{d}. \tag{5.8}$$

As mentioned in section 4.4, σ_{n} reflects, in this case, the degradation of performance due to centralized I/O strategy. Consider the case of a parallel machine connected via communication network to a gateway processor which in turn is connected to a RAID disks controller. Besides the time taken by the disks to manage the data, there are other overheads related to the path processors memory/disks. These overheads are: the time spent transferring data from the processor memories to the I/O processor, the time to transfer the data through the bus of the RAID controller and the time taken by the controller to strip the data across the disks. σ_{n} can be considered as the sum of the above times.

Upon substituting equation (5.6) and equation (5.8) into (4.3), we can evaluate the speedup $s(p,d)$.

5.2.1 Asymptotic analysis

By means of τ_{comp} and $\tau_{\mathrm{I}/O}$ it is possible to know the speedup behavior when $p\to\infty$ or $d\to\infty$.

Case $p\to\infty$
When $w=0$ the communication sub–network is composed only by delay stations. Then ($h(m)\simeq \ln m$):

$$\tau_{comp} \simeq \ln\frac{p}{c}\left[\frac{\sigma_{\mathrm{p}}}{p}\ln c + \sigma_{\mathrm{r}0} + \sigma_{\mathrm{r}}p^{\frac{1-r}{r}}\right]. \tag{5.9}$$

This means that, when $r>1$ and $\sigma_{\mathrm{r}0}=0$, for large values of p, the speedup tends to $1/\tau_{\mathrm{I}/O}$ (because $\tau_{comp}\to 0$). On the contrary, if $r=1$ or $\sigma_{\mathrm{r}0}\neq 0$ we get $\tau_{comp}\to\infty$, then the speedup tends to zero.

If $w > 0$ from (5.6) we have:

$$\lim_{p\to\infty} \tau_{comp}(p) = \lim_{p\to\infty} \sum_{i=1}^{p/c} \frac{1}{i} R(i, Z(p), D(p)). \tag{5.10}$$

Studying the convergence of the series in (5.10), when $w > 0$, it is possible to conclude that the asymptotic speedup is zero. Indeed, for large values of p we can write:

$$R(i, Z(p), D(p)) \simeq iD(p) = iw\sigma_{\mathrm{r}} g(p),$$

and then:

$$\begin{aligned} \tau_{comp}(p) &= \sum_{i=1}^{p/c} \frac{1}{i} R(i, Z(p), D(p)) \cong \sum_{i=1}^{p/c} w\sigma_{\mathrm{r}} g(p) = \\ &= \sum_{i=1}^{p/c} w\sigma_{\mathrm{r}} p^{\frac{1-r}{r}} = \frac{p}{c} + \frac{1}{c} w\sigma_{\mathrm{r}} p^{\frac{1}{r}}. \end{aligned}$$

From the last equation we have:

$$\lim_{p\to\infty} \tau_{comp}(p) = +\infty.$$

We conclude that the asymptotic speedup is zero.

Case $d \to \infty$
In this case the I/O burst time tends to σ_{n}, while the computation term remains constant. We conclude that:

$$s(p, \infty) = \frac{1}{\tau_{comp} + \sigma_{\mathrm{n}}}.$$

5.3 BUS–AIO model

The BUS-AIO model represents centralized I/O architectures and parallel applications characterized by asynchronous I/O.

Since in this case we do not have any I/O synchronization, we remove from the model the fork/join pair. The I/O burst is represented by a queueing station with service time given by $c(\sigma_{\mathrm{n}} + \sigma_{\mathrm{d}}/(dp))$ (see Figure 5.6). As in the optimistic AIO case, we assume that the service time of the I/O queueing station scales as $1/(dp)$. Furthermore, since c processors are involved in a collective communication, it is reasonable to assume that these processors perform the I/O burst at the same time. Therefore, we consider the service time of the I/O station proportional to c.

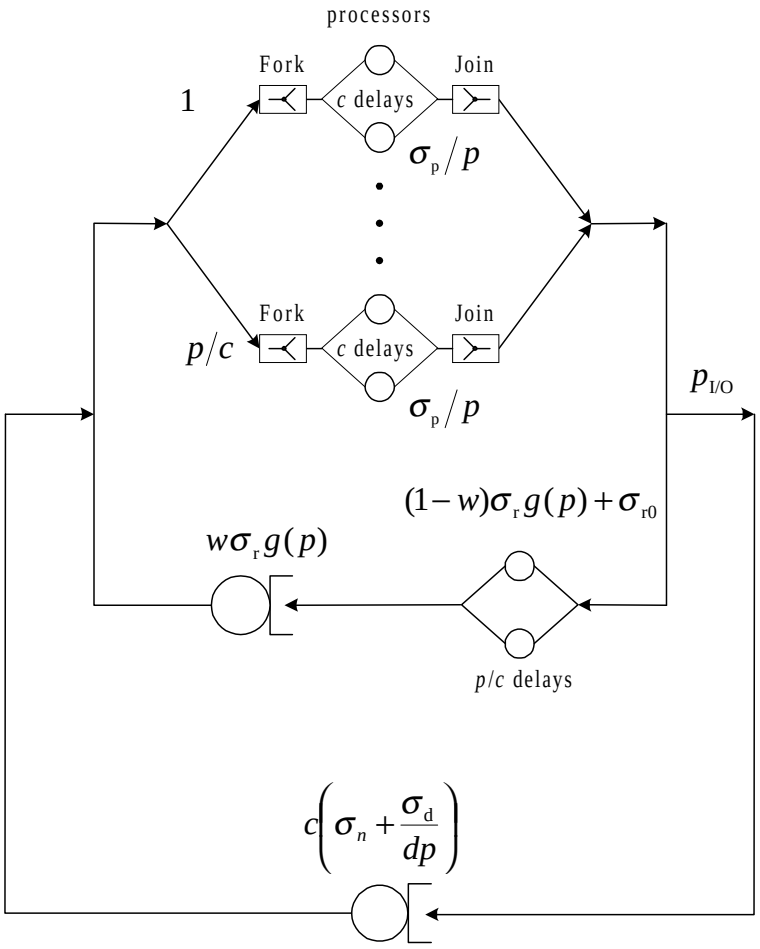

Figure 5.6: BUS–AIO queueing network model.

Though the two BUS models are similar, the solution technique is substantially different. BUS–AIO queueing network can be solved by means of exact MVA algorithm. This queueing network is composed by a delay station Z(p), given by (5.5) and by two queueing stations. The service time of the first queueing station is $D(p)$, given by (5.4). The service time of the second one is:

$$E(p,d) = c\left(\sigma_{\mathrm{n}} + \frac{\sigma_{\mathrm{d}}}{dp}\right). \tag{5.11}$$

By using MVA we have:

$$\tau_{comp}(p,d) = Z(p) + D(p)\frac{\displaystyle\sum_{(e_1,e_2,e_3)\in L^3_{p/c-1}} \frac{e_1+1}{e_3!} D(p)^{e_1} E(p,d)^{e_2} Z(p)^{e_3}}{\displaystyle\sum_{(e_1,e_2,e_3)\in L^3_{p/c-1}} \frac{1}{e_3!} D(p)^{e_1} E(p,d)^{e_2} Z(p)^{e_3}} \tag{5.12}$$

and

$$\tau_{\mathrm{I/O}}(p,d) = E(p,d)\frac{\displaystyle\sum_{(e_1,e_2,e_3)\in L^3_{p/c-1}} \frac{e_2+1}{e_3!} D(p)^{e_1} E(p,d)^{e_2} Z(p)^{e_3}}{\displaystyle\sum_{(e_1,e_2,e_3)\in L^3_{p/c-1}} \frac{1}{e_3!} D(p)^{e_1} E(p,d)^{e_2} Z(p)^{e_3}} \tag{5.13}$$

where L^3_p is the set of triples (e_1, e_2, e_3) of non–negative integers such that $e_1 + e_2 + e_3 = p$. From (5.12) and (5.13) we can evaluate the speedup of BUS–AIO model.

5.3.1 Asymptotic analysis

Case $p \to \infty$.
When $w = 0$ we have a simple queueing network composed by a single queueing station, $E(p,d)$, and a delay station, $Z(p)$. The asymptotic response time is roughly given by $pE(p,d) = pc\sigma_{\mathrm{n}} + c\sigma_{\mathrm{d}}/d$ and tends to infinite. We then conclude that the asymptotic speedup is zero.

When $w > 0$ we have two queueing stations with service time given by $D(p)$ and $E(p,d)$. We note that the response time of a closed queuing network with large population loads is dominated by the station with the greatest service time. In general, the asymptotic response time of a closed queueing network is given by $ND_{\overline{k}}$, where N is the population and $D_{\overline{k}}$ the greatest service time time. Since $E(p,d)$ comprises the constant term σ_{n}, we conclude that the asymptotic speedup is zero.

Case $d \to \infty$
$E(p,d) \to \sigma_{\mathrm{n}}$. Therefore the speedup tends to the reciprocal of the response time queueing network obtained by setting $E(p,d) = \sigma_{\mathrm{n}}$.

5.4 CLU–SIO model

In some architectures, clusters of processors share a common I/O node connected to a bus that is separated from the communication network. In this case, we consider only the configurations such that $p \geq d$. Indeed, the case in which $d > p$ is very rare: more than one I/O node for each processor.

In the CLU–SIO model, d clusters of p/d processors, sharing a common I/O node, perform I/O in synchronous manner. From the point of view of the modeling, BUS–SIO and CLU–SIO are identical and the only difference is in the meaning and the value of the parameter σ_{n}. The advantage of the CLU–SIO is that σ_{n} can be, in some cases, zero (i.e., the program does not perform any data redistribution before transferring).

5.5 CLU–AIO model

The model for the CLU–AIO is a multi class closed queueing network in which the job classes represent tasks running in different clusters of processors. The multi class technique allows the simulation of different groups of tasks, belonging to different clusters, following separate paths to the disks.

As mentioned, we consider only the case $p \geq d$. For sake of simplicity, we restrict our attention to cases in which $c \leq p/d$ and p/d multiple of c, which ensures that the number of processors synchronized before the communications belong to the same cluster. However, the methodology of analysis for the more general case without restriction on c is essentially the same.

The queueing model for the CLU–AIO has d job classes: a class represents a set of jobs that is executed in one of the d clusters of processors that shares a common I/O node. Each class has the same population $p/(cd)$.

Figure 5.7 depicts the CLU–AIO queueing network model. The vector of population $\overrightarrow{n}$ of dimension d, is given by:

$$\overrightarrow{n} = \left(\frac{p}{cd}, \frac{p}{cd}, \ldots, \frac{p}{cd}\right).$$

The service times of the queueing network is represented by a $(d + 2 \times d)$

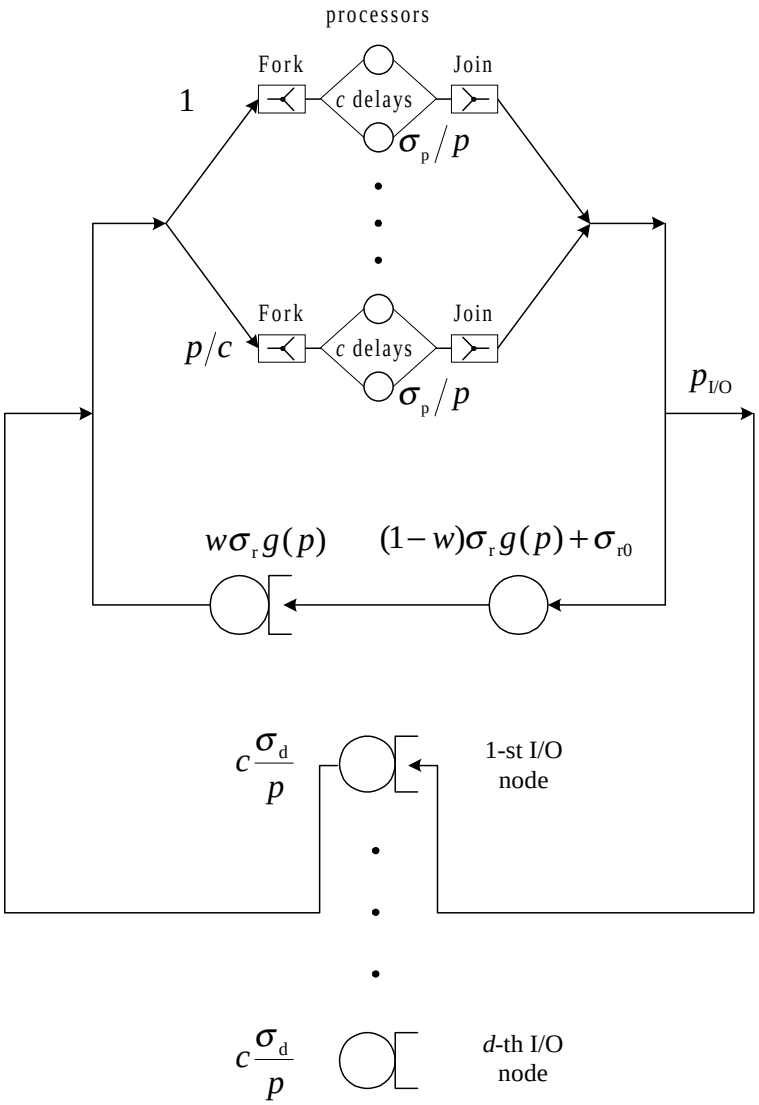

Figure 5.7: CLU–AIO queueing network model

matrix G given by:

$$G = \begin{pmatrix} c\frac{\sigma_{\mathrm{d}}}{p} & 0 & \dots & 0 \\ 0 & c\frac{\sigma_{\mathrm{d}}}{p} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & c\frac{\sigma_{\mathrm{d}}}{p} \\ D(p) & D(p) & \dots & D(p) \\ Z(p) & Z(p) & \dots & Z(p) \end{pmatrix},$$

where $D(p)$ and $Z(p)$ are defined in (5.4) and in (5.5), respectively. In order to solve the $CLU--AIO$ queueing network we use the MVA algorithm shown in section 3.2.2.

5.5.1 Asymptotic analysis

Case $p \to \infty$
When $w = 0$ the model can be considered as mono–class one with population $p/(cd)$, a delay $Z(p)$ and a queueing station with service time $c\sigma_{\mathrm{d}}/p$. The asymptotic response time of the queueing network is then given by:

$$\frac{p}{cd} \times c\frac{\sigma_{\mathrm{d}}}{p} = \frac{\sigma_{\mathrm{d}}}{d}.$$

Therefore, the asymptotic speedup (the reciprocal of the asymptotic response time) is d/σ_{d}.

When $w > 0$, we have two queueing stations. The asymptotic response time of the queueing network, is given by the product of the population $p/(cd)$, the number of classes d and the greatest service time. In this case the greatest service time is the one of the communication network station ($w\sigma_{\mathrm{r}}p^{\frac{1-r}{r}}$). The asymptotic response time of the queueing network is then given by $wd\sigma_{\mathrm{r}}p^{\frac{1}{r}}/c$. This means that the speedup tends to zero.

Case $p \to \infty$
Since we restricted our attention to the case $p \geq d$, the case $d \to \infty$ makes sense only if $p \to \infty$. This means that when $d = p$ and $d \to \infty$ the speedup tends to zero.

5.6 Case studies

5.6.1 Hybrid MIMD+SIMD machine

Suppose we have a MIMD machine boosted by massively parallel SIMD arrays, charged with the hugest number-crunching tasks. Consider a program executed on this hybrid machine. In this case, the SIMD processors

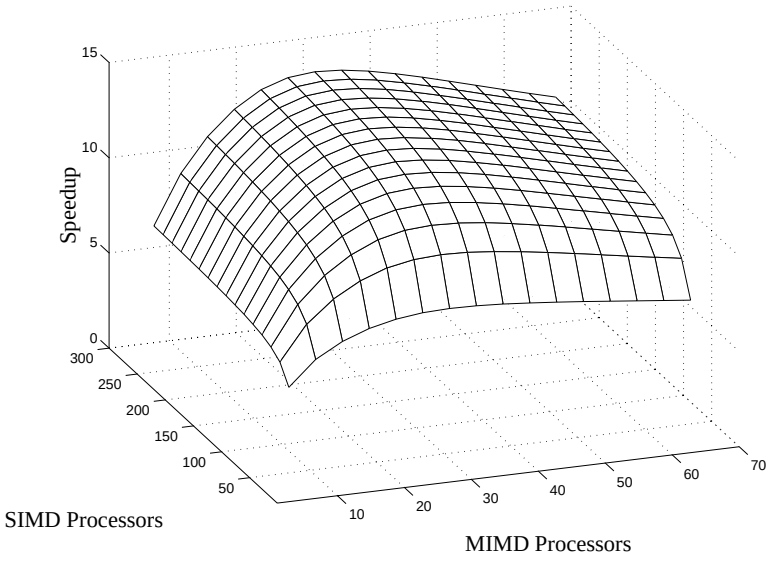

Figure 5.8: Example of the speedup surface of a program executed on a hybrid MIMD+SIMD parallel machine; $\sigma_{\mathrm{p}} = 0.2, \sigma_{\mathrm{r}} = 0.002, \sigma_{\mathrm{r0}} = 0.001, \sigma_{\mathrm{n}} = 0.02, r = 1, c = 1, w = 0.4$.

represent the I/O nodes of the system. We can study this scenario by means of the BUS–SIO model. We chose BUS–SIO since the SIMD board is connected to the MIMD machine according to the centralized structure. Moreover, we consider that the hybrid application performs the MIMD and the SIMD phase, in two disjoint intervals of time. Figure 5.8 depicts a possible speedup surface of a program executed on a hybrid machine with 64 MIMD and 256 SIMD processors. Note that a satisfactory speedup can be already obtained for $p = 28$ and $d = 128$.

5.6.2 BUS–AIO *vs* CLU–AIO

It is of interest to compare the performance of BUS and CLU strategies of parallel programs which perform out–of–core computations. In the former case, we suppose to have RAID disks. In the latter case, the disks are distributed among the processors according to the CLU architecture. The tradeoff between the two strategies is determined by the parameter σ_{n}, which is present only in the BUS model. We suppose, for instance, that the I/O burst of the program is 20% of the total execution time (i.e., $\sigma_{\mathrm{p}} = 0.8$). Figure 5.9 shows the speedups of BUS–AIO and CLU–AIO in the case of d= 4. We see that when $24 \leq p \leq 52$ the CLU–AIO case has the best speedup.

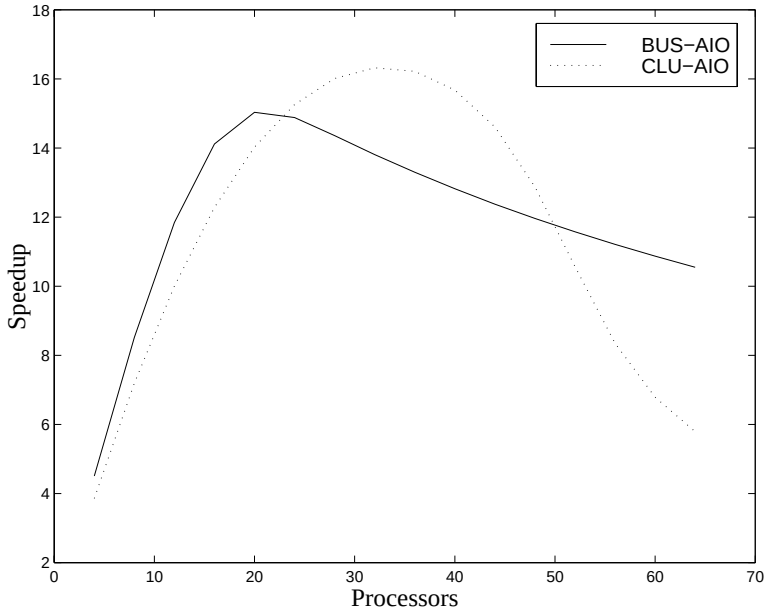

Figure 5.9: Speedup of a parallel program with different I/O strategies; $d = 4, \sigma_{\mathrm{p}} = 0.8, \sigma_{\mathrm{r}} = 0.005, \sigma_{\mathrm{r0}} = 0.001, \sigma_{\mathrm{n}} = 7 \cdot 10^{-4}, r = 1, c = 1, w = 0.2$.

Chapter 6

Program Behavior Results

In this chapter we will illustrate how the techniques developed in preceding sections can be used to study the behavior of real parallel applications. In section 6.1 a parallel implementation of MVA on a distributed memory machine, developed using the MPI library for communication, is described. We systematically investigate the performance of the algorithm and compare the observed speedup to the one obtained from the analytic model. The algorithm has been implemented on a Cray T3D machine.

In section 6.2 we infer the model parameters by fitting our model to an observed speedup surface obtaining the values that yield the best fit. The applications considered were selected among those of the Scalable I/O Initiative. Each application is comprised of a certain number of programs executed in a pipeline fashion, each of which is indicated as a "stage".

6.1 Parallel MVA algorithm

The Mean Value Analysis algorithm is one of the most popular for evaluating the performance of separable (or product form) queueing networks. Although its complexity is modest when jobs are indistinguishable, the introduction of different customer classes rapidly increases its computational cost. The problems of parallelizing the algorithm while retaining its conceptual simplicity are examined. In particular, a parallel implementation of MVA on a distributed memory machine is developed using the MPI library for communication [31].

Algorithms to numerically evaluate queueing networks have been the subject of much interest. Buzen[7] developed the first algorithm, known as the convolution algorithm. This algorithm finds the normalization constant G for a network of M centers and N customers using a simple recurrence relating $G(M, N)$ to $G(M-1, N)$ and $G(M, N-1)$. Other performance

metrics, such as mean queue lengths, centre utilizations, etc are found using G. Although very efficient, the convolution algorithm is not very intuitive, and its computations can be affected by overflow or underflow for large networks.

Reiser and Lavenberg[41] developed a new algorithm, Mean Value Analysis (MVA), that uses only meaningful metrics of network performance in its calculation.

A major advance in speeding up these algorithms has been the recognition of the tree structuring apparent when different classes of customer only visit subsets of the service centres. It is then possible to significantly simplify the calculations for those stations which are not visited by particular classes of customer. The full complexity of the algorithm needs to be applied only to service centers where different classes of customer interact. This simplification was originally discovered by Lam and Lien[32] and applied to the convolution algorithm. It can also be applied to MVA and RECAL.

Parallel implementations of a number of these algorithms have been proposed. Greenberg and McKenna[21] developed a parallel version of RECAL for use on shared memory multiprocessors. Pace and Tucci[38] worked with MVA. Greenberg and Mitrani[22] have developed a technique using fast Fourier transforms to evaluate the normalisation constant G in parallel. Hanson et al.[27] also used MVA.

Most of these algorithms have been implemented or proposed for a shared memory environment. It is a feature of all the algorithms, that they build up their solutions iteratively, either from the solutions of the same network with smaller populations, or from solutions to a smaller network with the same population. Shared memory means that earlier results are easily available on all processors.

Our interest is in the development of an algorithm which is effective in a distributed memory environment. Here each processor has its own storage, and data calculated on one processor is not available to other processors without explicit transmission to the other processor's memory. Inter processor communication needs to be minimized, because it will typically be several orders of magnitude slower than computations.

6.1.1 The algorithm

The aim of a parallel algorithm for MVA must be to calculate the same results as a uni-processor MVA algorithm, while gaining significant speedup by performing some of the calculation in parallel. The precedence graph will put an upper bound on the amount of parallelism that is possible.

We allocate a processor to be responsible for each population. Each pro-

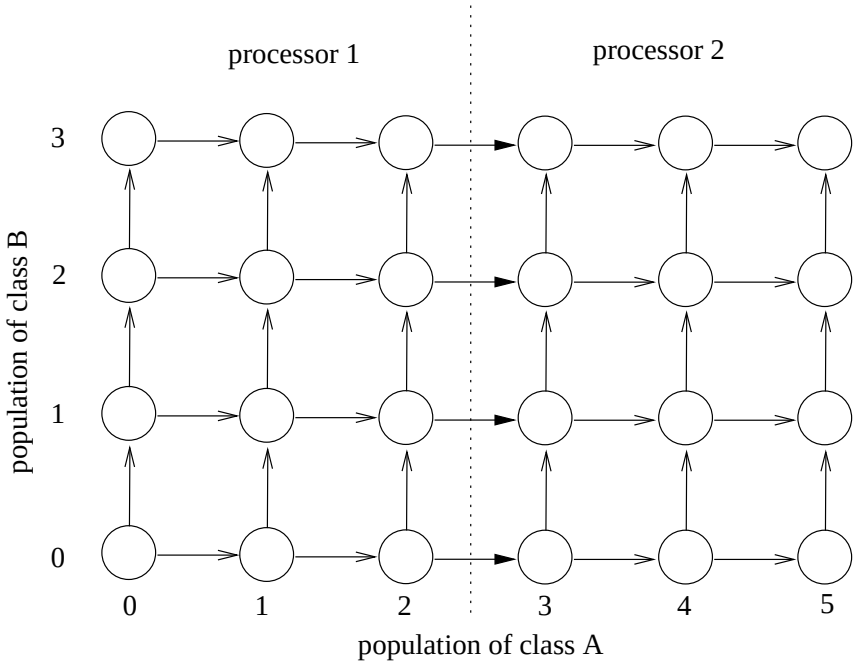

Figure 6.1: Example of the PMVA algorithm with 2 classes and 2 processors.

cessor may be allocated more than one population. As soon as the preceding populations have been calculated, calculations can start. If the preceding populations were allocated to different processors, then the performance vector must be transmitted between the processors. If the preceding population was calculated on the same processor, then no communication is needed.

Even if an unbounded number of processors were available, it would not be sensible to allocate only a single population to a processor. The communication cost in that case would overwhelm most of the speed up obtained by parallelism of the computation. We anticipate that a modest number of processors will achieve an almost linear speedup.

For simplicity, we allocate a population to a processor based only on the population of the first class of customers. This is easily implemented and gives a significant speed up. It might be possible to run some form of processor allocation algorithm which toured the precedence graph in order, allocating the population to a particular processor depending on whether the processor was already allocated, and on the identity of the processor used for neighboring populations in the graph.

6.1.2 The implementation

Figure 6.1 shows the precedence graph of computation in the case of queueing network with two customer classes (named A and B). Class A has a population of 5 customers and class B, 3 customers. The node at coordinates (r, s) corresponds to the computation of the statistics (queue lengths, response times, etc.) when the network has r customers of class A and s customers of class B. The calculation ends when the node at coordinates $(5, 3)$ has been evaluated.

In the case of a two processors implementation, the nodes are partitioned solely on the basis of the population of class A customers. Processor 1 calculates those nodes that have class populations from 0 to 2 inclusive, and processor 2 calculates those with class A populations of 3 or more. In general, when the final population is (m, n) processor 1 is allocated populations (i, j) for $0 \leq i \leq k$ and processor 2 is allocated $k < i \leq m$, where $k = m/2$. In Figure 6.1, $k = 2$ and processor 1 starts alone and executes the computation at nodes $(0, 0)$, $(1, 0)$, $\ldots$, $(k, 0)$, in order. It then sends the results of node $(k, 0)$ to processor 2, which has been idle until this time. Processor 2 executes the nodes from $(k+1, 0)$ to $(m, 0)$, and simultaneously processor 1 executes the nodes from $(0, 1)$ to $(k, 1)$. A pipeline is established with processor 1 is executing node (r, s) with $0 \leq r \leq k$ and simultaneously processor 2 computes the results for $(t, s-1)$ with $k+1 \leq t \leq m$. Eventually, processor 1 reaches node (k, n), calculates the performance for that population and transmits it to processor 2. Processor 1 is then idle while processor 2 works on nodes $(k+1, n) \ldots (m, n)$. When processor 2 reaches the node (m, n) and executes the corresponding computations, the algorithm terminates.

When more processors are available, the nodes are still partitioned among processors on the basis of their class 1 population. If a total class population of m customers is to be calculated, and there are p processors available, then each processor is assigned m/p values of class 1 population.

Networks with C classes of customer generate an C-dimensional precedence graph. Although more complex processor assignment algorithms would be possible, we have extended the two dimensional algorithm. The nodes are partitioned on the basis of the population of class 1 customers. Processor 1 starts by evaluating the nodes for populations of class 1 from 0 to k, while all other classes have populations of 0. When population k is reached, processor 2 starts with population $k+1$. Meanwhile, processor 1 has started the calculation of results for a population of 1 in class 2, again taking the class 1 population from 0 to k. When it reaches k, processor 2 should have finished computations for class 1 populations up to m, and should be ready to calculate for population $k+1$ again, but with a

population of 1 in class 2.

6.1.3 Performance prediction of the algorithm

Letting N_c be the population of class c for $1 \leq c \leq C$, the execution time $T(1)$ of the algorithm on a single processor is given by:

$$T(1) = q(1) \prod_{c=1}^{C} (N_c + 1), \tag{6.1}$$

where $q(1)$ represents the mean time spent computing a node of the MVA algorithm in the case of $p = 1$. Since the parallel machines exploit cache mechanisms during the computations, we assume that q may depend on the number of processors. The execution time $T(p)$ with $p > 1$ is given by:

$$T(p) = T_{cl}(p) + T_{cm}(p), \tag{6.2}$$

where $T_{cm}(p)$ is the time spent communicating and $T_{cl}(p)$ the time spent computing. The term $T_{cl}(p)$ can be estimated from the time to calculate one node, the number of processors, and careful accounting for the periods when not all processors are active. Each processor is responsible for a range of class 1 subscripts. Most processors deal with a range of size $\left\lceil \frac{N_1+1}{p} \right\rceil$ subscripts. The processors that are responsible for a smaller range will be idle for the processing of a single node. If we assume that the calculation of a single node takes $q(p)$, then the first processor will take time $q(p) \cdot \left\lceil \frac{N_1+1}{p} \right\rceil$ to calculate while processor 2 is idle. It will then proceed to calculate for the remaining populations of class 2 through to class C. Hence processor 1 will be busy for a time given by:

$$q(p) \left\lceil \frac{N_1 + 1}{p} \right\rceil \prod_{i=2}^{C} (N_i + 1)$$

Before the computation is complete, the pipeline must empty. This involves the remaining $p - 1$ processors each calculating for a time of $q(q) \left\lceil \frac{N_1+1}{p} \right\rceil$. Adding these terms we get

$$T_{cl}(p) = q(p) \left\lceil \frac{N_1 + 1}{p} \right\rceil \left(\prod_{i=2}^{C} (N_i + 1) + p - 1 \right) \tag{6.3}$$

This formula will be a slight overestimate if the processors are not all responsible for the same number of subscripts. Dividing them equally, if $N_1 + 1$ is not exactly divisible by p, one should give $\lceil (N_1 + 1)/p \rceil$ to processors 1,2,3,..,k, and one fewer subscripts to processors $k + 1, \ldots, p$. The

correction term for the pipeline emptying will be slightly smaller than that given above when the later processors have fewer subscripts for which to calculate.

In a similar manner, the communication time can be derived, assuming that all communications are synchronous.

$$T_{cm}(p) = k(p)\left(\prod_{i=2}^{C}(N_i+1)+p-2\right), \tag{6.4}$$

where $k(p)$ represents the mean time spent communicating the queue lengths of a node of the MVA algorithm from a processor to its successor in the pipeline when the number of processors is p[1]. Finally we can give the expression of the total time $T(p)$ by means the equations (6.2), (6.3) and (6.4):

$$T(p) = q(p)(N_1+1)\left(\frac{1}{p}\prod_{i=2}^{C}(N_i+1)+\frac{p-1}{p}\right)+k(p)\left(\prod_{i=2}^{C}(N_i+1)+p-2\right) \tag{6.5}$$

6.1.4 Experimental results

The pipelined implementation described above has been implemented on the Cray T3D machine at the Edinburgh Parallel Computing Centre. The algorithm was restricted to the case of load independent service centers. This was only for implementation convenience, and does not represent a restriction on the applicability of the method. The C programming language was used, with all real values being expressed as `double` variables, which occupy 8 bytes. The MPI message passing library was used to provide interprocessor communications. Synchronous communication was used, so that the sender of a message would block until it had been successfully received.

The following parameters were chosen so that the program takes about 30 minutes elapsed time when running on a single processor.

- $C = 3$ (number of classes);
- $K = 5$ (number of centers);
- $N_1 = 4095, N_2 = 177, N_3 = 127$ (classes populations).

The execution times[2], the speedups and the relative efficiency, of the program with increasing number of processors are shown in the Figures 6.2, 6.3 and 6.4, respectively.

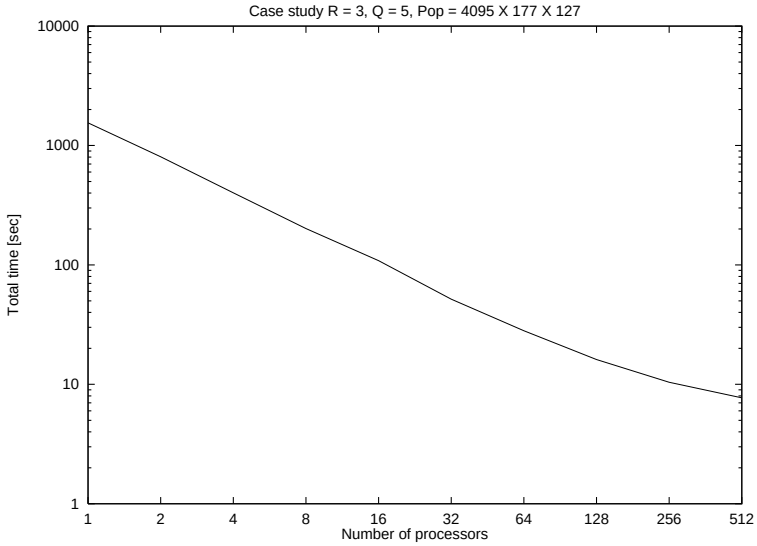

Figure 6.2: Execution times of the pipeline MVA algorithm on Cray T3D machine.

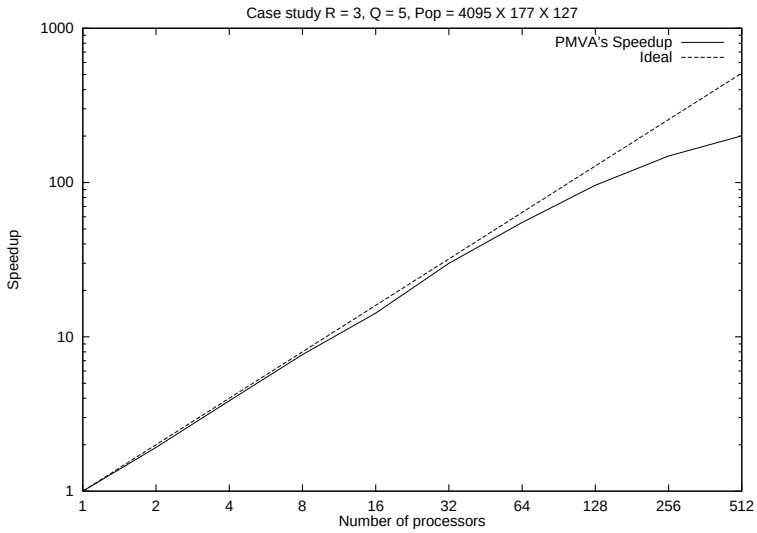

Figure 6.3: Speedup of the pipeline MVA algorithm on Cray T3D machine.

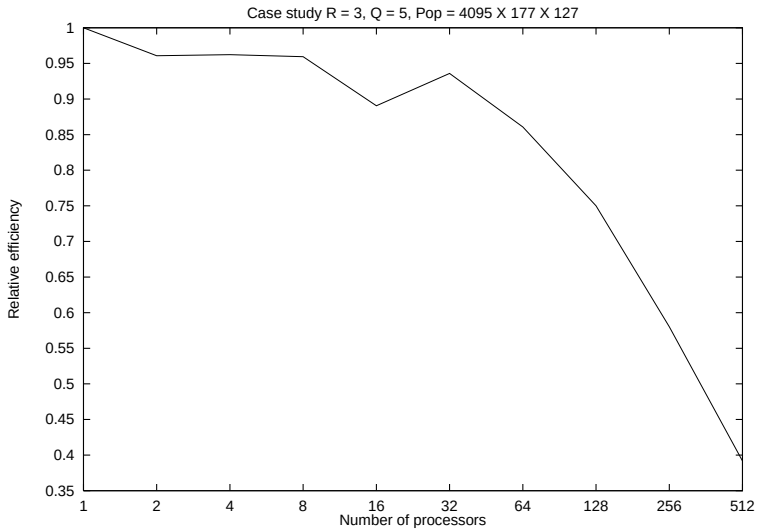

Figure 6.4: Relative efficiency of the pipeline MVA algorithm on Cray T3D machine.

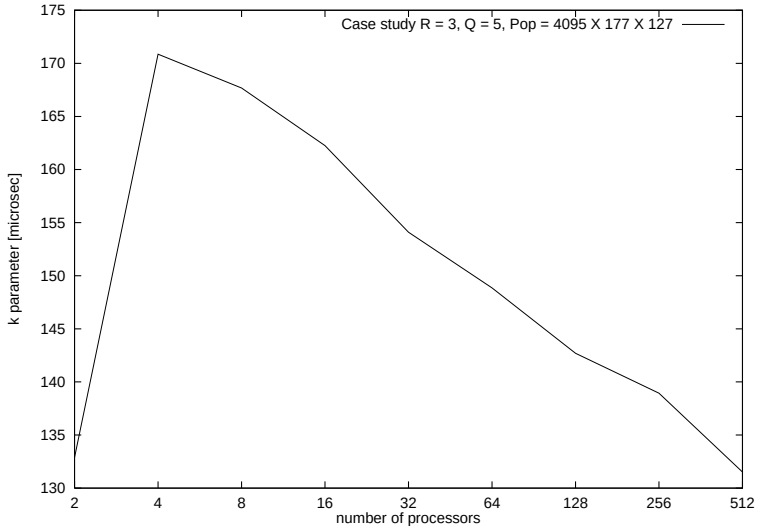

Figure 6.5: The k parameter of the pipeline MVA algorithm on Cray T3D machine.

Figure 6.5 shows the average $k(p)$ with p from 2 to 512. Notice the difference between $k(2)$ and $k(p)$ when $p > 2$. This is due to the different behavior of the pipeline and the use of synchronous communications among the processors (see Figure 6.6). Further when $p > 2$ the mean time communicating $k(p)$ decreases as the number of processors increases. This is because the relative amount of load imbalance decreases as the amount of work is distributed amongst more processors decreases. Suppose a processor finishes a lot sooner than all the others, it will start wait until the receiver is ready to receive, hence its communication time will reflect this. As the amount of work is distributed any potential load imbalance will decrease in proportion, hence it will appear that the communication time is coming down.

6.1.5 Interpretation of the results

Using Equation (6.1) and the experimental results we can evaluate the parameter $q(1)$:

$$q(1) = \frac{T(1)}{\prod_{i=1}^{R}(N_i + 1)} = 16.6\mu sec. \tag{6.6}$$

In order to evaluate $q(p)$, when $p > 1$, we can exploit the measured time $K_i(p)$ that is the total time spent communicating from the processor i. Since the measured time $T(p)$ can be considered as the execution time of p-th processor, we obtain:

$$q(p) = \frac{p(T(p) - K_p(p))}{\prod_{i=1}^{C}(N_i + 1)} = 16.6\mu sec^{3}. \tag{6.7}$$

Figure 6.7 shows the parameter $q(p)$ obtained in this way. We observe that the computation time $q(p)$ depends on the number of processors. This effect is because of differing data being stored in the high speed memory cache.

[1] Notice that $k(p)$ includes the waiting time of the synchronous communications.

[2] Because the T3D machine does not accept job running with only one processor, the execution time for the case $p = 1$ is considered to be equal the time obtained with the program running on two processor job with population of class 1 given by $2(N1+1)-1$, without communications and keeping one of two processors idle.

[3] The time $K_i(p)$ also includes the time spent waiting in the communication operations.

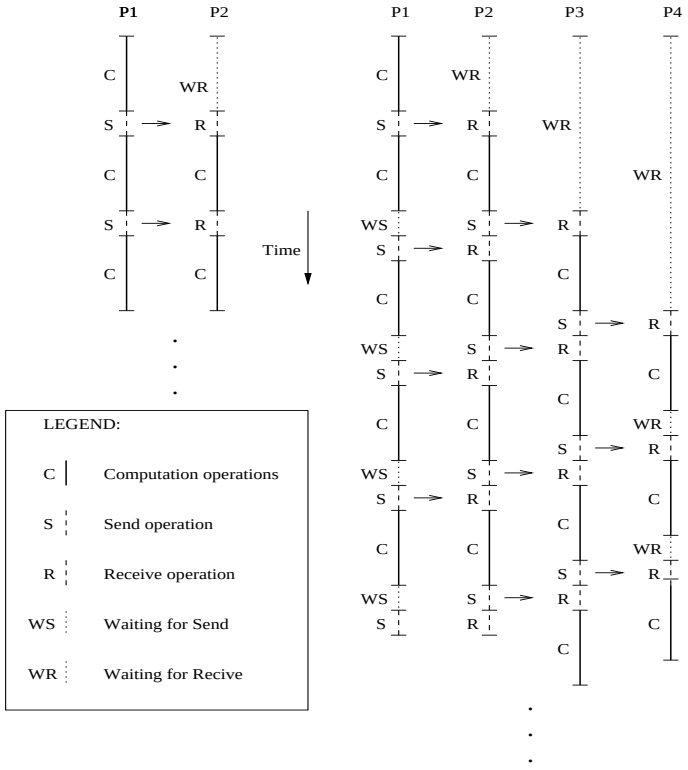

Figure 6.6: Behavior of the pipeline MVA algorithm with two and four processors.

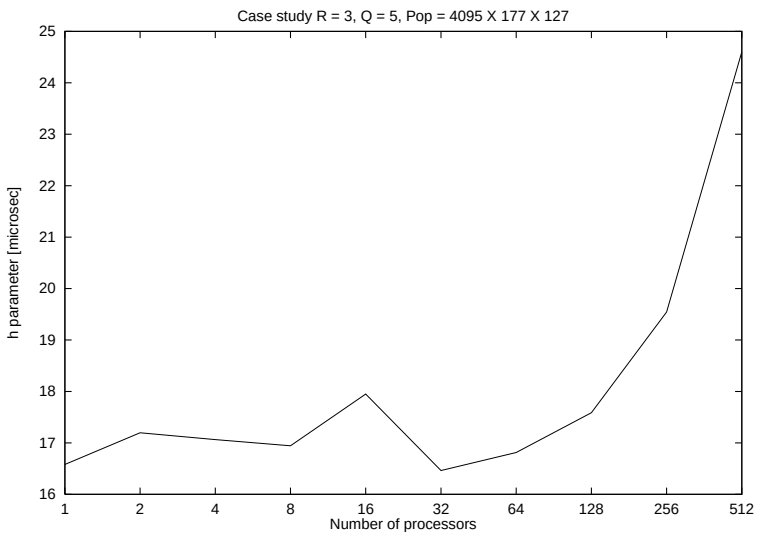

Figure 6.7: The q parameter of the pipeline MVA algorithm on Cray T3D machine.

6.1.6 Improving the performance of the program

The performance of the pipeline MVA algorithm depends strongly on the input problem. The speedup $s_{mva}(p)$ of the program is given by dividing $T(1)$ by $T(p)$, that is:

$$s_{\mathrm{mva}}(p) = \frac{q(1)\prod_{i=1}^{C}(N_i+1)}{q(p)(N_1+1)\left(\frac{1}{p}\prod_{i=2}^{C}(N_i+1)+\frac{p-1}{p}\right)+k(p)\left(\prod_{i=2}^{C}(N_i+1)+p-2\right)}. \tag{6.8}$$

Dividing both numerator and denominator of (6.8) by $k(p)$ and supposing $\alpha(p) = \frac{h(p)}{k(p)} \approx \frac{h(1)}{k(p)}$[4], we obtain:

$$s_{\mathrm{mva}}(p) = \frac{\alpha(p)\prod_{i=1}^{R}(N_i+1)}{\alpha(p)(N_1+1)\left(\frac{1}{p}\prod_{i=2}^{R}(N_i+1)+\frac{p-1}{p}\right)+\prod_{i=2}^{R}(N_i+1)+p-2}. \tag{6.9}$$

[4]This is true when we can neglect the effect of the cache. In our case study this is true when $p \leq 256$.

When the effect of the pipeline delays are negligible, that is when $\prod_{i=2}^{R}(N_i + 1) \gg p$, Equation (6.9) becomes:

$$s_{mva}(p) = \frac{\alpha(p)(N_1+1)}{\frac{\alpha(p)}{p}(N_1+1)+1}. \tag{6.10}$$

Then the relative efficiency $e(p) = \frac{s_{mva}(p)}{p}$ is:

$$e(p) = \frac{\alpha(p)(N_1+1)}{\alpha(p)(N_1+1)+p}. \tag{6.11}$$

Hence when $\alpha(p)(N_1 + 1) \gg p$ we can obtain from the program a relative efficiency near to 1.

Equation (6.11) implies that one should arrange the classes such that class 1 has the largest population. This will ensure that $\alpha(p)(N_1 + 1)$ is as large as possible with respect to p. The effect of the ordering of the other classes will be minimal, although cache effects may give rise to small differences in performance.

Figure 6.8 shows the execution times when the same queueing network is analyzed, but with the classes presented in a different order. For instance the case ABC means the class A corresponds to the class with index $i = 1$ in the MVA algorithm, the class B with index $i = 2$, and so on. We see that the two cases ABC and ACB have the best execution times, that is when the class with the largest population has index $i = 1$.

6.1.7 Model validation

In order to evaluate the usefulness of our speedup models, we compare the performance figures of the pipeline MVA with our analytic model result.

Because pipeline MVA does not perform any I/O, the type of models CLU and BUS are, in this case, equivalent. We must choose the AIO model, since, even though there is not I/O ($\sigma_p = 1$) we do not want that the processors are synchronized at the end of the computation burst. The absence of I/O can be obtained by setting $\sigma_p = 1$ (i.e., $\sigma_d = 0$) and $\sigma_n = 0$.

Since we have synchronous communications between all pairs of contiguous processors of the pipeline, we choose $c = 2$. Moreover, there is no contention using the communication network, because the pipeline stages are mapped to contiguous processors of the T3D mesh. Therefore, $w = 0$.

The algorithm data space is unidimensional ($r = 1$), and hence $g(p) = 1$.

In [14] performance measures of MPI send (standard and synchronous) are available. The bandwidth T_b and the latency L, for the message size

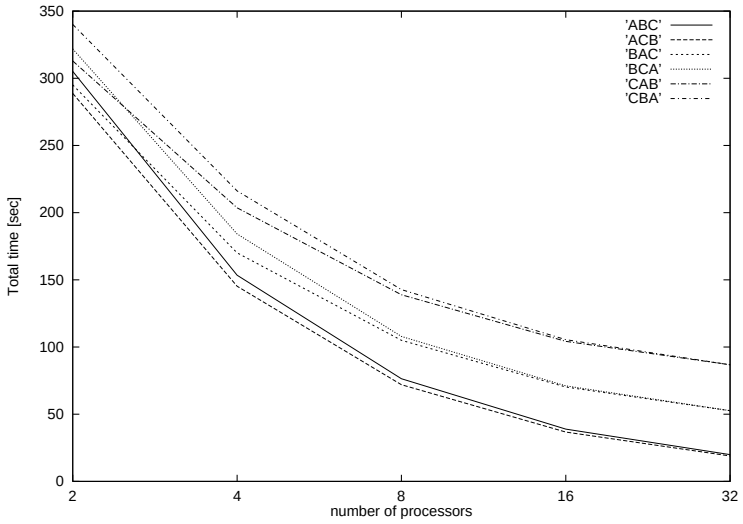

Figure 6.8: Execution time of the PMVA with different order of classes in the algorithm. $N_A = 4095$, $N_B = 127$, $N_C = 63$.

$M_s = 40$B of our case study, are 115MB/sec and 46.9μsec, respectively. In order to evaluate σ_r and σ_{r0} we need to know to computation time of the PMVA algorithm between two communications. This is the average time spent computing a node of the MVA algorithm ($\simeq q(1) = 16.6\mu sec$), multiplied the population of class 1 ($N_1 = 4095$). By normalizing the communication times, with respect to the computation time, we obtain:

$$\sigma_r = \frac{M_s}{T_b q(1)(N_A + 1)} = 5.3074 \cdot 10^{-6}, \tag{6.12}$$

$$\sigma_{r0} = \frac{L}{q(1)(N_A + 1)} = 7.1564 \cdot 10^{-4}. \tag{6.13}$$

Figure 6.9 shows a comparison between the experimental speedup with the one obtained from the BUS–AIO model. Note that we needed only the sequential computation time and an estimate of the communication times.

6.2 Speedup surfaces of applications with intensive I/O

In this section we show the speedup surfaces of three I/O intensive programs. The applications considered were selected among those of the Scal-

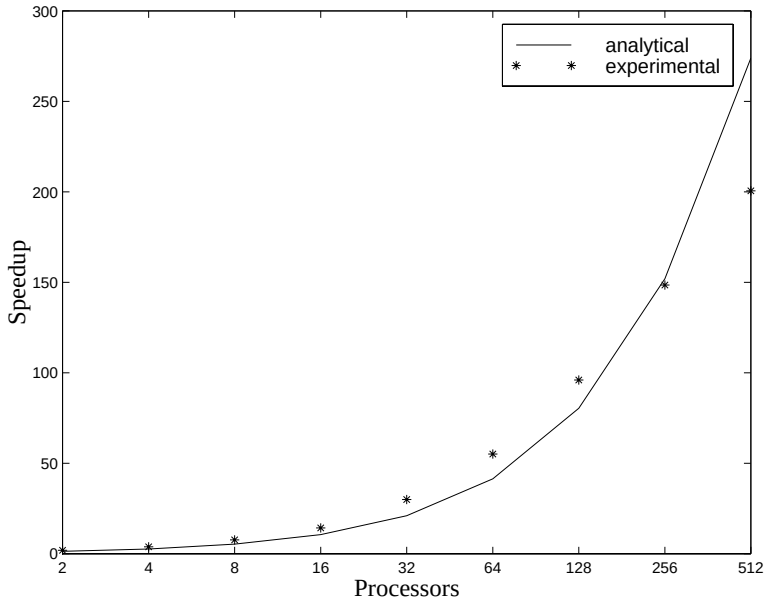

Figure 6.9: Comparison between the experimental speedup and BUS–AIO model speedup.

able I/O Initiative[5], that exhibit a nonnegligible speedup. Each application is comprised of a certain number of programs executed in a pipeline fashion, each of which is indicated here as a "stage" [42].

The experimental platform used is the 512-node Intel Paragon XP/S with 64 4GB Seagate disks, each attached to a computation node, at the Caltech Center of Advanced Computing Research. Performance measures are collected using Pablo, a performance analysis environment that provides trace data for the I/O and CPU requests of the parallel applications.

The first application considered is QCRD (Quantum Chemical Reaction Dynamics) that solves the Schroedinger equation for the differential and integral cross section of the scattering of an atom by a diatomic molecule. It was effectively used to investigate the reaction between a deuterium atom and a diatomic hydrogen molecule and led to the discover of a geometric phase effect that had not been detected before. QCRD implements the method of symmetrical hyperspherical coordinates and local hyperspherical surface functions with a typical SPMD structure. All nodes execute the same code on different portions of the data set each of equal size so as

[5]The Scalable I/O Initiative is an effort to collect a suite of I/O intensive national challenge scientific applications, characterize their behavior in terms of I/O access patterns, analyze their performance and use the gathered information to design and evaluate policies for the management of parallel file systems.

to keep the load balanced. The execution is divided into 5 consecutive stages that proceed in a pipeline fashion. During stage 1, a twodimensional eigenvalueeigenvector problem is solved and a primitive surface function is obtained. All the nodes perform a series of interleaved writes to the basis file. During stage 2, each node independently computes a subset of the integrals that are needed to evaluate the two-dimensional quadratures involving the primitive basis functions. Both read and write operations are performed by all nodes. In stage 3, the integrals computed at the previous stage are collected into dense matrices. Each node reads the basis matrices and solves the tridiagonal system resulting from the tridiagonalization of the generalized eigensystem. Different parts of the Hamiltonian matrices that were solved for are stored at different nodes. In stage 4 read operations are followed by computation and then write operations. Every interaction matrix is loaded at each processor in order to compute the collision energies. Once the propagator file is computed, it is written out to disks by all nodes. Finally, in stage 5 the propagator file is read by all nodes, which then compute the scattering matrices and write the results to the disks. We focus on stage 2 and stage 4 because they achieve reasonable speedups.

The second application is MESSEKIT, an electronic structure calculations application that uses the Hartree–Fock algorithm. The electronic density around a molecule is computed by considering each of the molecule electrons in the collective field of the others, iterating the computation until the field felt by each electron is consistent with that of the other electrons. Input to the algorithm is the basis functions derived from the atoms and the relative geometries of the atom centers. The Coulomb interactions between electrons are computed by solving the atomic integrals over the basis functions, thus producing an approximate molecular density. The density and the atomic integrals are then used to derive a Fock matrix. A selfconsistent field (SCF) method is finally applied until the molecular density converges to within an acceptable threshold. The application is comprised of three logical stages executed in a pipelined fashion. During the first stage, called setup, a single node, node 0, is active reading initialization data from disk, calculating the basis sets and writing the results out to disk. Such a stage is not parallel so it is not modeled here. In stage 2, called ARGOS, all nodes participate in the computation of the atomic integrals, each node writing a private file with the locally computed integrals. The granularity of the I/O and CPU bursts is fine. During stage 3, called SCF, all nodes repeatedly read the integral data, construct the Fock matrix, compute, synchronize, and solve the SCF equations. All nodes operate concurrently, with node 0 that periodically reads the intermediate results and writes them out to disk. We focus on stage 2.

Since in the algorithm of the ARGOS and QCRD stage 2 the proces-

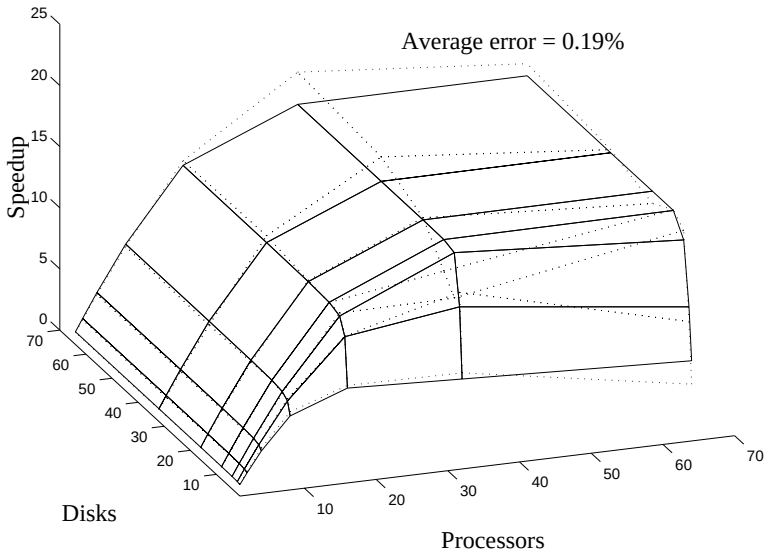

Figure 6.10: ARGOS – Experimental speedup surface (dashed line) *vs* analytical (solid line); $\sigma_{\mathrm{p}} = 0.8805, \sigma_{\mathrm{r0}} = 0.0070, \sigma_{\mathrm{r}} = 0.0605, r = 1.4055 \cdot 10^{5}, w = 0.9455, \sigma_{\mathrm{n}} = 5.1883 \cdot 10^{-4}$.

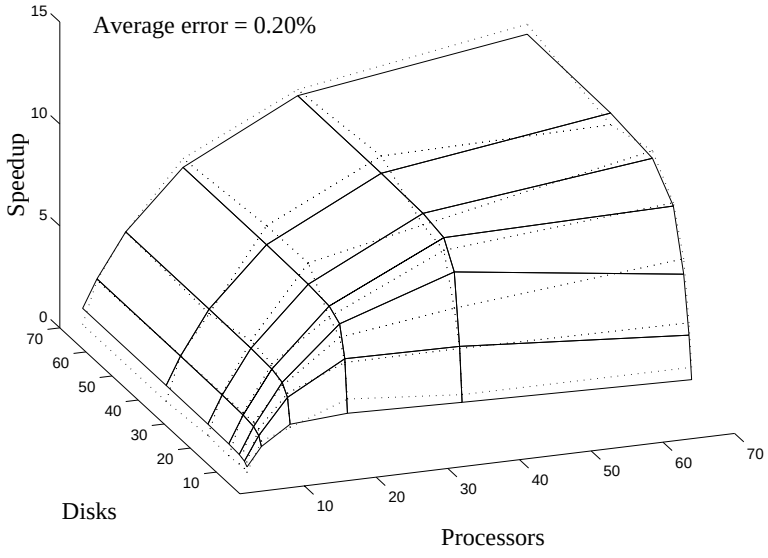

Figure 6.11: QCRD stage 2 – Experimental speedup surface (dashed line) *vs* analytical (solid line); $\sigma_{\mathrm{p}} = 0.7118, \sigma_{\mathrm{r0}} = 0.0487, \sigma_{\mathrm{r}} = 0.4125, r = 4.5296 \cdot 10^{12}, w = 0.1871, 9.0 \cdot 10^{-4}$.

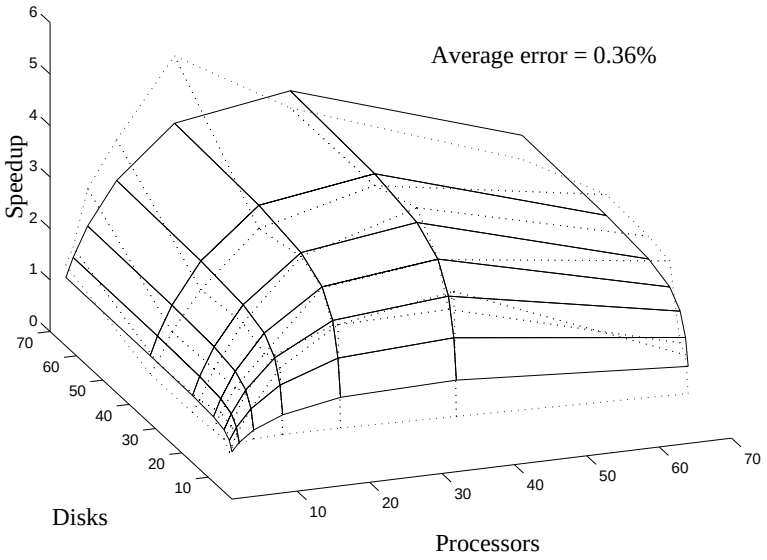

Figure 6.12: QCRD stage 4 – Experimental speedup surface (dashed line) *vs* analytical (solid line); $\sigma_{\rm p} = 0.6585, \sigma_{\rm r0} = 0.0, \sigma_{\rm r} = 0.0013, r = 0.6985, w = 0.4260$.

sors perform read/write operation independently, we chose to use the model BUS–AIO. Instead, for the QCRD stage 4 the BUS–SIO is more appropriate.

Given a set of observed speedup $s_{p,d}$ we estimated values of the model parameters that minimize the sum of squared differences between the observed values and the fitted values $s(p,d)$. In other words, we minimized:

$$\epsilon = \sum_{p,d} \left[s(p,d) - s_{p,d} \right]^2 .$$

Figure 6.10, 6.11 and 6.12 show the speedups observed *vs* the speedups obtained from the estimate parameters[6]. In each case, we observe that the fitted model is a good match for the observed data. Note that in ARGOS and in QCRD stage 2 the value of r tends infinite. This means that the communication time scale as $g(p) = 1/p$.

[6]Note that the average error is defined as:

$$\sum_{p,d} \frac{\left[s(p,d) - s_{p,d} \right]^2}{s_i^2}.$$

Chapter 7

Conclusion

In this thesis we have described a modeling approach for investigating parallel programs performance when executed on different types of systems. From these models we obtain the qualitative and quantitative behavior of programs that alternate computations and I/O in a cyclic fashion. The proposed performance model allows to study the impact of both the communication contention and I/O of the system, showing the dependence between speedup and number of processors and I/O nodes in the parallel machine. Various aspects of the communication and I/O have been analyzed and different hardware architectures have been taken into consideration.

The purpose of this thesis is to permit an estimate of the program speedup by inserting in the models the computation times, the communication times, the number of processors involved in the synchronous communications and the number of dimensions of the program space data. Concerning the I/O we need to know the size of the data and an estimation of the parameter σ_{n}. The communication contention level w is the most difficult to determine, however, by assigning $w = 1$ and $w = 0$ it is possible to obtain a lower bound and a upper bound of the speedup, respectively.

Moreover, we have shown that we can infer the program characteristics of a program by fitting our model to observed speedup. With the proposed modeling technique, the fitting of experimental speedup surfaces produced very small errors; thus it would be appropriate to use these estimated parameters for allocation and scheduling.

Bibliography

[1] G. M. Amdhal. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS 1967 Spring Joint Computer Conference*, volume 30, pages 483–485, April 1967.

[2] F. Baccelli and Z. Liu. On the execution of parallel programs on multiprocessor system – a queuing theory approach. *Journal of the Association for Computing Machinery*, 37(2):373–414, 1990.

[3] K. R. Backer. *Introduction to Sequencing and Software.* John Wiley & Sons, 1974.

[4] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.

[5] S. Baylor and C. Wu. *I/O in parallel and distributed computer systems*, chapter Parallel I/O worload characteristics using Vesta, chapter 7, pages File–access character. Kluwer Academic Publisher, 1996.

[6] J. P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, September 1973.

[7] J.P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, September 1973.

[8] W. W. Chu and K. K. Leung. Module replication and assignment for real–time distributed processing systems. In *Proceedings of the IEEE Vol. 75 No. 5*, 1987.

[9] E. G. Coffman and P. J. Denning. *Operating System Theory.* Prentice–Hall, N. J. Henglewood Cliff, 1973.

[10] P. Cremonesi and C. Gennaro. I/o performance in hybrid mimd+simd machines. In *Proceedings of High-Performance Computing and Networking 98*, volume 1017 of *Lecture Notes in Computer Science*, Amsterdam, April 1998. Springer-Verlag.

[11] R. Suros E. Gelenbe, E. Montagne and C. M. Woodside. A performance model of block–structured parallel programs. In *Proceedings of the International Workshop on Parallel Algorithms and Architectures*, pages 127–138. North–Holland, 1986.

[12] T. Philips E. Gelenbe, R. Nelson and A. Tantawi. The asymptotic processing time for a model of parallel computation. In *Proceedings of the National Computer Conference*, Las Vegas, USA, 1986.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Speedup versus efficiency in parallel systems. *IEEE Transaction on Computers*, 38(3):408–423, 1989.

[14] EPCC. Mpi for t3d performance measures, jun 1997. Available at http://www.epcc.ed.ac.uk/t3dmpi/Product/Performance/.

[15] G. Fayolle, P. J. B. King, and I. Mitrani. On the execution of programs by many processors. In *Proceedings of Performance 88*, pages 217–228. North–Holland, Amsterdam, 1983.

[16] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1–20, 1989.

[17] I. Foster. Design and building parallel programs, 1995. Available at www.mcs.anl.gov/dbpp/text/.

[18] E. Gelenbe. *Multiprocessor Performance*, pages 83–90. John Wiley & Sons, Series in Parallel Computing, 1989.

[19] E. Gelenbe and Z. Liu. Performance analysis approximation for parallel processing in multiprocessor systems. In *Proceedings of the IFIP Working Conference on Parallel Processing*, pages 363–375. North–Holland, 1988.

[20] C. Gennaro. Performance models for i/o bound spmd applications on clusters of workstations. To appear on *Proc. of 7th Euromicro Workshop on Parallel and Distributed Processing*, 1999.

[21] A.G. Greenberg and J. McKenna. Solution of closed, product form, queueing networks via the RECAL and tree-RECAL methods on a shared memory multiprocessor. *Performance Evaluation Review*, 17(1):127–135, 1989.

[22] A.G. Greenberg and I. Mitrani. Massively parallel algorithms for network partition functions. In *International Conference on Parallel Processing*, Chicago, January 1991.

[23] J. L. Gustafson. Reevaluating amdahl's law. *Communication of the ACM*, 31(5):532–533, 1988.

[24] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. and Stat. Computing*, 9(4):609–638, 1988.

[25] J.L. Gustafson. The scaled–sized model: A revision of amdhal's law. *ICS Supercomputing*, II:130–133, 1988.

[26] K. Salem H. Garcia-Molina. The impact of disk-striping in reliability. *IEEE Database Engineering Bulletin*, 11(1):26–33, mar 1988.

[27] F.B. Hanson, J.-D. Mei, C. Tier, and H. Xu. PDAC: A data parallel algorithm for the performance analysis of closed queueing networks. *Parallel Computing*, 19(12):1345–1358, 1993.

[28] U. Herzog and W. Hoffmann. Syncrhonization problems in hierachically orgasnized multiprocessor computer systems. In *Performance of Computer System, Proc. 4th Int. Symp. Modeling Performance Evaluation Computer Syst.*, pages 29–48. North–Holland, Amsterdam, 1979.

[29] A. N. Choudhary J. M. del Rosario. High–performance i/o for massively parallel computers. *IEEE Computer*, pages 59–68, March 1994.

[30] J.R. Jackson. Jobshop-like queueing systems. *Manage. Sci.*, 10(1):131–142, 1963.

[31] P. J. B. King and C. Gennaro. Parallelising the mean value analysis algorithm. To appear on Special Issue on Parallel and Distributed Simulation *Transaction of the Society for Computer Simulation*, 1999.

[32] S.S. Lam and Y.L. Lien. A tree convolution algorithm for the solution of queueing networks. *Communications of the ACM*, 26(3):203–215, March 1983.

[33] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance – Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[34] J. D. C. Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, pages 9:383–387, 1961.

[35] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing 91*, pages 567–576, November 1991.

[36] P. Mussi and J. T. Nain. Evaluation of parallel execution of program tree structures. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 78–87, 1984.

[37] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. File–access characteristics of parallel scientific workloads, technical report pcs–tr95–263. Technical report, March 1995.

[38] L. Pace and S. Tucci. A parallel algorithm for distributed computer performance evaluation environments. In *Proc. 1990 Summer Computer Simulation Conference*, pages 797–802, 1990.

[39] B. K. Pasquale and G. Plyzos. A static analysis of i/o characterization of scientific applications in a production workload. In *Proceedings of Supercomputing 93*, pages 388–397, November 1993.

[40] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[41] M. Reiser and S.S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):313–322, 1980.

[42] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante. The impact of i/o on program behavior and parallel scheduling. In *ACM Sigmetrics Conference*, June 1998.

[43] P. J. Schweitzer. Exact solution of the MVA equations. *SIAM Review*, 23:528–532, 1981.

[44] K. S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall, Durham, North Carolina, 1982.

[45] M. Vanneschi. Pqe2000: Hpc tools for industrial applications. *IEEE Concurrency*, pages 68–71, oct–dec 1998.

[46] K. Wilson. High–performance computing and communications, grand challenges 1993 report. Technical report, Report by the Committee on physical, mathematical and engineering sciences federal coordinating council for science engineering and technology, Washington D.C., 1993.

[47] X. Wu and W. Li. Performance models for scalable cluster computing. *Journal of System Architecture*, 44:189–205, 1998.

Appendix A

Non–recursive MVA formula

In this appendix we derive a non–recursive form of the MVA algorithm for queueing networks with one job class.

Consider the queueing network shown in Figure A.1. Let S_j $(1 \leq j \leq k)$ and Z denote the service time of the j–th queueing stations and the delay of the terminal, respectively. Let p be the number of jobs in the network. The mean response $R_j(p)$ time of the j–th queueing center is given by:

$$R_j(p) = S_j \left[\frac{\displaystyle\sum_{(e_1,\ldots,e_{k+1}) \in L_{p-1}^{k+1}} \frac{e_j}{e_{k+1}!} S_1^{e_1} \ldots S_k^{e_k} Z^{e_{k+1}}}{\displaystyle\sum_{(e_1,\ldots,e_{k+1}) \in L_{p-1}^{k+1}} \frac{1}{e_{k+1}!} S_1^{e_1} \ldots S_k^{e_k} Z^{e_{k+1}}} \right] \tag{A.1}$$

where L_n^m is the set of m–tuple $(e_1, \ldots, e_m)$ of non negative integers such that $e_1 + \ldots + e_m = n$.

We show a proof of (A.1) for $k = 1$, a complete proof can be find in [43].

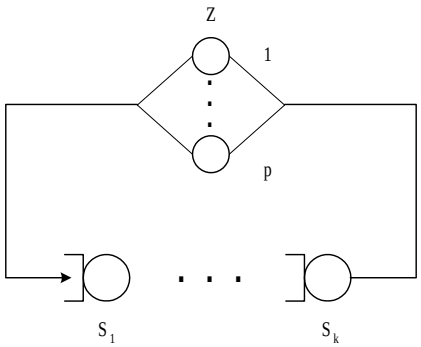

Figure A.1: Queuing network used for non–recursive MVA solution.

Proof. By induction on p. When $p = 1$ it is straightforward to verify that (A.1) holds. Assume that (A.1) holds for $p-1$, we show that it holds for p. From (A.1) we can write $R(p-1)$ as:

$$R(p-1) = S\left[\frac{N(p-1)}{D(p-1)}\right] \tag{A.2}$$

Where

$$N(p) = \sum_{i=0}^{p-1} \frac{p-i}{i!} S^{p-i-1} Z^i$$

and

$$D(p) = \sum_{i=0}^{p-1} \frac{1}{i!} S^{p-i-1} Z^i$$

The response time $R(p)$ is given by [33]:

$$R(p) = S\left[Q(p-1) + 1\right] \tag{A.3}$$

Where $Q(p-1)$ is the mean queueing length when in the network are $p-1$ jobs circulating. Let $X(p)$ be the throughput of the jobs, given by:

$$X(p) = \frac{p}{Z + R(p)}$$

Therefore using the Little's law [34] ($Q = RX$) and (A.3):

$$R(p) = S\left[\frac{(p-1)R(p-1)}{Z + R(p-1)} + 1\right]$$

Upon substituting (A.2) into the last equation we obtain:

$$R(p) = S\left[\frac{(p-1)SN(p-1)}{ZD(p-1) + SN(p-1)} + 1\right] = S\left[\frac{pSN(p-1) + ZD(p-1)}{ZD(p-1) + SN(p-1)}\right] \tag{A.4}$$

Substituting the expressions of $N(p-1)$ and $D(p-1)$ into (A.4), we get the following formula:

$$R(p) = S\left[\frac{\displaystyle\sum_{i=0}^{p-2} p\frac{p-i-1}{i!} S^{p-i-1} Z^i + \sum_{i=0}^{p-2} \frac{1}{i!} S^{p-i-2} Z^{i+1}}{\displaystyle\sum_{i=0}^{p-2} \frac{1}{i!} S^{p-i-2} Z^{i+1} + \sum_{i=0}^{p-2} \frac{p-i-1}{i!} S^{p-i-1} Z^i}\right] \tag{A.5}$$

In (A.5) it is straightforward to collapse the terms $S^j Z^k$ of the polynomials with the same degrees j and k. For instance, for the numerator:

the generic term i–th of the left polynomial (i.e., $p\frac{p-i-1}{i!}S^{p-i-1}Z^i$) added to the term $(i-1)$–th of the right polynomial (i.e., $\frac{1}{(i-1)!}S^{p-i-1}Z^i$), gives $\frac{p(p-i-1)+i}{i!}S^{p-i-1}Z^i = (p-1)\frac{(p-i)}{i!}S^{p-i-1}Z^i$. Finally, we obtain:

$$R(p) = S\left[\frac{\displaystyle\sum_{i=0}^{p-1}(p-1)\frac{p-i}{i!}S^{p-i-1}Z^i}{\displaystyle\sum_{i=0}^{p-1}(p-1)\frac{1}{i!}S^{p-i-1}Z^i}\right] = S\left[\frac{\displaystyle\sum_{i=0}^{p-1}\frac{p-i}{i!}S^{p-i-1}Z^i}{\displaystyle\sum_{i=0}^{p-1}\frac{1}{i!}S^{p-i-1}Z^i}\right] \qquad \text{(A.6)}$$

Appendix B

Fork/join queueing network response time

In this section we evaluate the average response time $R_{F/J}$ of the fork/join system of Figure B.1.

The response time of this system can be seen as the time taken by the p jobs to exit the queueing network. We use the Markov chain of Figure B.2 in order to obtain an approximate value of this time.

Let α and U be the exit probability and the utilization of the queueing station, respectively, the exit rate is given by $\alpha U/S$, from which we obtain the time $R_{F/J}$:

$$R_{F/J} = \sum_{i=1}^{p} \frac{S}{\alpha U_i}, \tag{B.1}$$

where U_i is the utilization of the queue for population i and S is service time of the queueing station. From the Little's law we know that:

$$U_i = X_i S = \frac{i}{R_i} S \tag{B.2}$$

where X_i and R_i are the respective throughput and response time of the queueing network for the population i. Substituting (B.2) into (B.1) we have:

$$R_{F/J} = \frac{1}{\alpha} \sum_{i=1}^{p} \frac{R_i}{i} \tag{B.3}$$

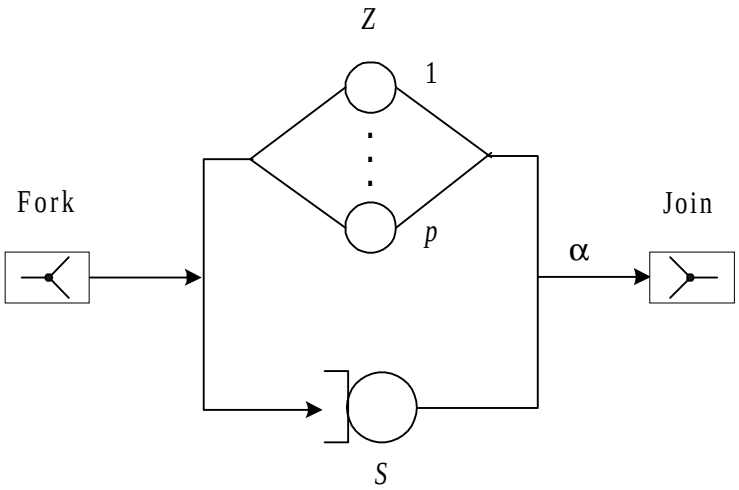

Figure B.1: fork/join system: α = exit probability, S = queue service time, Z = terminal think time, p = population.

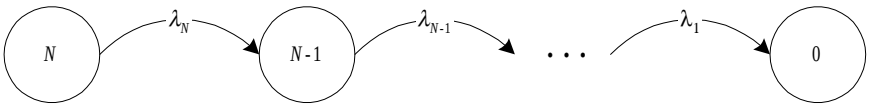

Figure B.2: Markov chain used for evaluating the average fork/join response time.