NATIONAL INSTITUTE OF ELECTRICITY AND ELECTRONICS
INELEC - BOUMERDES

DEPARTMENT OF RESEARCH

THESIS

Presented in partial fulfillment of the

DEGREE OF MAGISTER

in Electronic Systems Engineering

by

CHEREF MOHAMED

TASK PARTITION BASED PARALLEL DESIGN AND IMPLEMENTATION OF THE MSSM ALGORITHM ON A NETWORK OF TRANSPUTERS

Defended on July 03, 1995 before the jury

Registration Number 04/95

ACKNOWLEDGMENTS

I like to express my most sincere gratitude to my technical advisor Mr. A. BOUKLACHI for his continued research guidance that without his advises I would not have finished this research work.

I am specially grateful to Dr. K. HARICHE, head of the research department, for providing research facilities to make our project a success.

My special thanks to Mr. A. BOUCHOUATA, Mrs. BENTOUMI, Miss. F. BOUZIDI, and Mrs. BERRI.

DEDICATION

To mounia

ABSTRACT

The present work is concerned with the parallel design and implementation of the Multiple Scale Signal Matching (MSSM) [26,38] algorithm on a transputer [7,10,25,40] network. The MSSM algorithm is based on a multichannel vision model [35], to establish the correspondence between two images with the allowance that one of them can be deformed elastically.

The MSSM algorithm uses a process consisting of two stages: the filtering and the matching stages. This process is iterated following a coarse-to-fine regime of the vision channels [35] at which the matching process is performed.

Therefore, the algorithm exhibits a certain degree of computational complexity over huge amounts of data. This suggests the use of parallel processing to reduce the execution time of the algorithm. For this reason we have considered the parallelization of the algorithm over a PC transputer network with the OCCAM 2 [4,18,28,41] language under the Transputer Development System TDS3 [19]. A task partition approach is used to parallelize the algorithm. First, the algorithm is partitioned into a set of elementary tasks. Then, an intertask data flow is established. Afterwards, the network topology is fixed and subsequently the tasks are placed onto the transputer network processors. Finally, the tasks are scheduled onto processors in order to minimize the processors idle time caused by the intertask data flow, and obtaining an optimal starting time for each task ignition.

The experiments were carried out to measure the performance of the parallel implementation with respect to the sequential one. The effect of the increase in the computational time when additional vision channels are used, with respect to the increase in the number of processors has been also investigated. The performance tests indicate a substantial improvement in speed compared with a single processor execution. The performance analysis has permitted us to identify the optimal number of processors suitable for such an application.

CONTENTS

Introduction 1

CHAPTER 1

1- Motivation for multiprocessor and parallel processing systems

1-1 Introduction 5

1-2 Acceleration mechanism at the realization level 6

1-3 Acceleration mechanism at the implementation level 7

1-3.a Memory Latency 9

1-3.b Memory Bandwidth 10

1-3.c Execution Latency 12

1-3.d Execution Bandwidth 12

1-4 Acceleration mechanism at the architectural level 15

1-4.a Special Purpose Architecture 15

1-4.b Risc Architecture 15

CHAPTER 2

2- Parallel systems taxonomies

2-1 Introduction 18

2-2 Reasons for architectural classification 19

2-3 Flynn's classification 19

2-3.a SISD Computers 20

2-3.b MISD Computers 20

2-3.c SIMD computers 21

2-3.d MIMD computers 22

2-4 Treleaven's classification. 23

2-4.a COSH computers 24

2-4.b COME computers 25

2-4.c DASH computers 25

2-4.d DAME computers 25

2-4.e DESH computers 25
2-4.f DEME computers 26
2-4.g PASH computers 26
2-4.h PAME computers 26
2-5 Skillicorn's classification 27

CHAPTER 3

3- Parallel processing systems programming

3-1 Introduction 30
3-2 Shared memory systems programming 31
3-3 Distributed memory systems programming 33
3-4 Task allocation scheduling 35
3-4.a Graph theoretical approach 36
3-5 Performance measure 40

CHAPTER 4

4- Transputers

4-1 Introduction 42
4-2 Transputer architecture 43
4-3 Transputer instruction set 46
4-4 The OCCAM language 47
4-5 The transputer development system 53
4-5.a INMOS SPRINT board description 54
4-5.b INMOS SPRINT board configuration 55
4-6 Transputer family 57

CHAPTER 5

5- The Multiple Scale Signal Matching algorithm

5-1 Introduction 60
5-2 General description 61
5-3 The filtering stage 62
5-4 The matching stage 69

CHAPTER 6

6- Sequential design and implementation of the MSSM algorithm

6-1 Introduction 72
6-2 Program design 73
6-3 The interface process. 75
6-4 The MSSM process 81
6-4.a Receiver process 82
6-4.b Task process 83
6-4.c Sender process 84
6-5 Conclusion. 85

CHAPTER 7

7- Parallel design and implementation of the MSSM

7-1 Introduction 87
7-2 Algorithm decomposition 88
7-3 Parallel program design 88
7-3-1 Topology 1 (two transputers) 89
7-3-1.a Task allocation for n=1. 89
7-3-1.b Task allocation for n=2. 91
7-3-1.c Task allocation for n=3. 95
7-3-1.d Task allocation for n=4. 96
7-3-2 Topology 2 (Three transputers) 97
7-3-2.a Task allocation for n=1 97
7-3-2.b Task allocation for n=2 97
7-3-2.c Task allocation for n=3 99
7-3-2.d Task allocation for n=4 99
7-3-3 Topology 3 (Four transputers). 100
7-3-3.a Task allocation for n=1. 100
7-3-3.b Task allocation for n=2. 101
7-3-3.c Task allocation for n=3. 101
7-3-3.d Task allocation for n=4. 102
7-4 Implementation. 102
7-5 Results. 110
CONCLUSION. 114
REFERENCES

Introduction :

Recently, much research interests have been devoted to the development of new acceleration mechanisms for computer systems in order to meet the application computational needs such as the ones concerned with the field of computer vision (image understanding or scene analysis). Many acceleration techniques have been proposed and have ceased to be efficient just a while before being used. This fact motivated computer scientists towards multiprocessing concepts. The multiprocessing concepts allow an application program to be run on a parallel computer in a parallel fashion. A parallel computer consists of a collection of processing units (i.e. processors, that cooperate to solve a problem by working simultaneously on different parts of it). As a result, the time required to solve the problem by a traditional uniprocessor computer has been significantly reduced.

The major distinction between the many proposed parallel systems is the fact of being or not being based on the shared memory scheme. The *shared memory* system consists of a set of conventional *CPU's* that are connected to a common memory through a single bus. The communication between the different CPU's is achieved by the shared memory. However, since all data traffic has to take place over the same bus, this later will be saturated as soon as a handful number of processors is used. Hence, the performance of the system is slowed down in terms of execution speed. The non-shared memory systems which are referred as *distributed memory* systems consist of a set of processors that have their own private memory. In the d[illegible]ibuted memory system, the processors can be connected through a static or dynamic interco[illegible]ection network such as a programmable cross bar switch. The eventual communication bet[illegible] the processors can be established by message passing. However, this kind of communication scheme has required a special hardware interface that obeys to a strict communication policy.

On the other hand, there have been many challenging problems of quite tremendous computational complexities. For instance, computer vision requires many digital image processing techniques[44] whose operations are computationally huge and act on large dimensional image arrays. Moreover, most computer tasks possess a large amount of inherent parallelism so that they can be split down into smaller tasks and allocated according to a certain criterion processing policy on hardware processors of a parallel system.

The algorithm that will be implemented in this research work has been introduced in the field of computer vision and is called the Multiple Scale Signal Matching (MSSM) [26,38]. It is based on the multichannel model [35] of human vision. The MSSM algorithm deals with the matching of two images where one is used as the reference. The matching process is based on the elastic matching theory [46]. The input to the MSSM algorithm consists of two globally aligned images and the output is under the form of a horizontal and a vertical discrepancy map. The discrepancy maps represent the measure of images discrepancy. They contain the corresponding coordinates of each pixel of the matched image with respect to the reference one.

This thesis is concerned with the parallel design and implementation of the MSSM algorithm on a distributed memory system using transputers [7,10,25,40]. The distributed memory systems are more advantageous than the shared memory systems because they are scalable. The idea here is that the performance of the system is maintained by an eventual use of a large number of processors. As this type of systems requires the provision of a well adapted hardware for the different processors to communicate with each other, new processors have been introduced such as transputers and Wrap processor [16]. In addition, suitable programming languages have been dedicated to such systems as well such as the OCCAM language [4,18,28,41].

In the design of the parallel MSSM algorithm, a task allocation scheduling [30,43] approach is employed. The parallel design proceeds by first decomposing the program into a set of elementary tasks which are then mapped on the hardware processors so that an optimal task allocation scheduling will be obtained. The procedure of the task allocation scheduling can be considered as a function which maps the elementary tasks from a sequential space onto a parallel one. An optimal amount of parallelism can be obtained if to each task that is mapped from the sequential space to the parallel one, a best starting point in time of its ignition is acquired on the parallel space.

The algorithm is implemented on a transputer network using a set of INMOS SPRINT boards [33]. The programming environment is based on the INMOS Transputer Development System (TDS3) [19] and OCCAM. TDS3 manages the resources of the parallel system.

The thesis will be presented in the following chapters:

In Chapter 1, a discussion on the motivation that led computer designers to turn to parallel processing systems will be initiated through an examination of the different acceleration mechanisms that have been proposed at the different levels of computer system design.

In Chapter 2, we will review some of the important computer architecture taxonomies.

In Chapter 3, parallel systems programming issues will be described with a particular stress on distributed systems and task allocation scheduling techniques.

In Chapter 4, we will review the transputer since it is the basic computing element in our design and implementation hardware. The hardware, software and development system will be described.

In Chapter 5, we will give descriptions of the different stages of the MSSM algorithm.

In Chapter 6, we will present the sequential design and implementation of the MSSM algorithm on a single transputer.

In Chapter 7, we will discuss the parallel design and implementation of the MSSM algorithm based on a task partition approach followed by the performance measure of the parallel implementation with respect to the sequential one.

Finally, we conclude with remarks and further scope.

CHAPTER 1

MOTIVATION FOR MULTIPROCESSOR SYSTEMS AND PARALLEL PROCESSING

1-1 Introduction:

Since the appearance of the computer, the demand for higher computer system performance has never ceased. To face such a situation, computer designers have always been looking for new acceleration design techniques through which higher performances could be achieved.

In response to the complexity of computer systems, the design process has been divided into three distinct levels [3]: the realization level related to the technology and type of material used for the implementation of the different computer system components, the implementation level that is concerned with the organization and the interconnection topology of the several components of the machine, as well as the specification of rules governing the flow of data and control signals between them, and finally the architectural level dealing with the principle of the computer basic function operations; such as arithmetic and logic operations.

The mechanisms used to accelerate computer system are classified according to the three design levels at which they are applied. An overview of the computer system acceleration techniques is shown in Figure 1.1.

This chapter describes the different acceleration techniques proposed and introduces the idea that motivated computer designers toward parallel systems and the parallel processing approach.

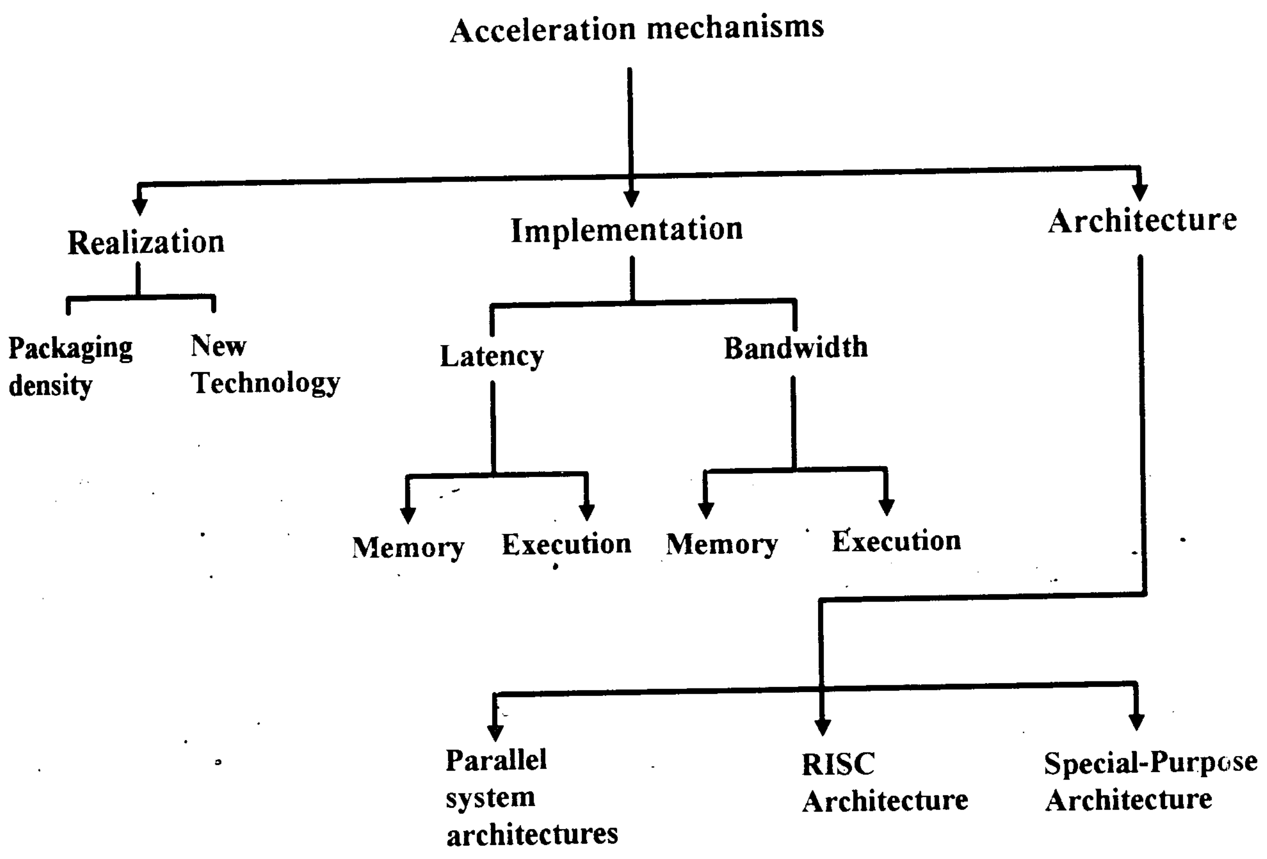


Figure 1.1: Computer system acceleration mechanisms.

1-2 Accelerations at the realization level:

At the realization level, the speed of the computer system is related to the technolog[illegible]f the devices or the components used at the implementation level to construct the des[illegible]d computer system architecture.

The first computer generation was realized with electromechanical relays. In the [illegible]t generation, these relays were replaced by vacuum tubes which were superseded later by transistors. In the following generation, the transistors were packaged into small-scale integrated (SSI), and medium-scale integrated (MSI) circuits. In addition, magnetic core memories were replaced by solid state memories.

Due to the increasing packaging density, the technological improvements have resulted into large-scale integration (LSI), and very large scale integrated (VLSI) chips that are commonly used for the implementation of commercialized computers. For example, a recent microprocessor that uses this high packaging density, the MOTOROLA MC68040 [12] integrates a MC68030 CPU, a 66882 math coprocessor, memory management unit, and a local cache memory on a single chip. The current trend is directed towards achieving a higher density known as Wafer-Scale Integration (WSI) in which a set of processors can be integrated separately but interconnected on a wafer to build, for instance a multiprocessor system.

Another way used to improve the computer system performance at the realization level is the search for a new type of material which can offer higher speed than the one used. For example, Gallium Arsenide (GaAS) offers a higher speed than Silicon (Si), as it can withstand higher temperatures.

This new technology and specially the high density packaging has given an evolutionary step in the speed up improvement of the computer system by producing less propagation delays and having high speed circuits used for the implementation of the several components of the computer system. However, due to the physical limitations, the possibilities for the acceleration at this level have almost reached their limits.

1-3 Accelerations at the implementation level:

At this level, the design process is concerned with the organization of the different units of the computer system, the topology of their interconnection, and the rules governing the flow of data between them. To have a better view of the different acceleration techniques at this level, it is useful to consider the basic von-Neumann machine organization as shown in Figure 1.2.

Basically, the von-Neumann machine operates as follows: the CPU *fetches* the instruction from memory, *decodes* it, and finally *executes* the instruction and stores the result in a proper location.

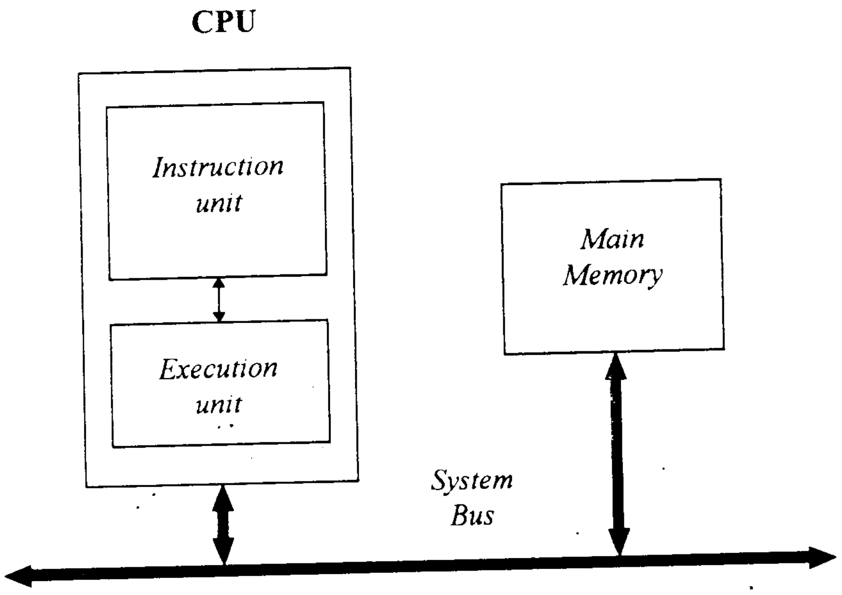


Figure 1.2 : Simple computer system units organization.

From the above description, four issues that are related to the speed of the computer can be identified and are defined as follows:

1- *Memory latency:* The time period (time delay) of the memory response.

2- *Memory bandwidth:* The amount of data that can be transferred per second from and to memory.

3- *Execution latency:* The time taken for an instruction to be executed.

4- *Execution bandwidth:* The amount of data processed per second.

It can be noted that the computer speed up at the implementation level is inversely proportional to the latency aspect of both memory and execution units, and inversely proportional to also both memory and execution bandwidth. In order to achieve computer speed up improvements some form of acceleration mechanisms are needed to reduce the memory and execution latency (time delay), and others to increase both the memory and execution bandwidth (time rate).

1-3.a Memory latency:

The first technique that has been used to decrease the memory latency is based on the fact that the access time of the registers (5 to 10 ns) is less than the access time of the main memory (250 to 1000 ns). Therefore, memory latency could be reduced by increasing the number of registers so that most frequently used operands can be stored and accessed faster than if they were stored in the main memory. Most microprocessors use this technique, such as the MC68000 which possesses 16 working registers (8 data register, 8 address registers). However, this technique tends to be avoided due to the fact that, the chip area is expensive and limited. For example this technique is not used in the design of the INMOS transputers [7,10,25,40], where only six working registers are available.

Another way used to reduce the memory latency is by using high speed memories. Presently, very fast memories are designed with an access time around 50 ns. They are placed between the CPU and the main memory (slow memory) and are called Cache memories as shown in Figure 1.3.

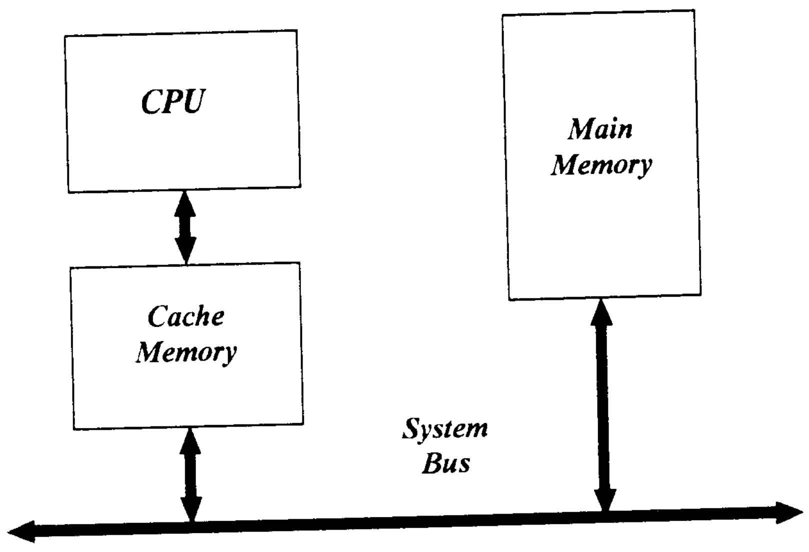


Figure 1.3: Computer system organization with cache memory.

As the Cache memories are very expensive, only a small amount of fast memories are used. They are combined with a large amount of slow memories. The most frequently referenced data must be transferred from large slow memory (main memory) into the small fast memory, so that it can be accessed more quickly.

1-3.b Memory bandwidth:

One way used to increase the memory bandwidth is based on the pipelining approach. This approach consists of dividing the main memory into several modules. Each module responds to the processor's memory request independently. This technique is referred as *memory interleaving* [3].

The idea on which this technique is based is that when higher processing power is to be achieved, larger and faster memories are required. However, larger memories require larger address decoding time resulting in lower memory bandwidth. To solve such a problem, the main memory is partitioned into many separate modules. If the main memory is divided into M separate modules, m (=Log_2 M) bits will be required to activate one of the M modules. The remaining bits are used to address a word from the activated module.

Using this technique, two approaches can be applied. The first approach is called lower-order interleaving. It is called so, because the lower order bits are used to select one of the modules in which the required word is to be found (Figure 1.4). The second approach is called higher order interleaving. The higher order bits of the address are used to select one of the modules in which the required word is stored (Figure 1.5).

The low order interleaving approach has the advantage to support spatial locality [3]. In other words consecutive addresses are found or stored in consecutive modules. Since instructions are executed in a sequential manner, the consequence of storing them in consecutive modules makes this technique the most frequently used. This organization is well suited for accessing vector elements when data are represented by vectors as in the case of image processing.

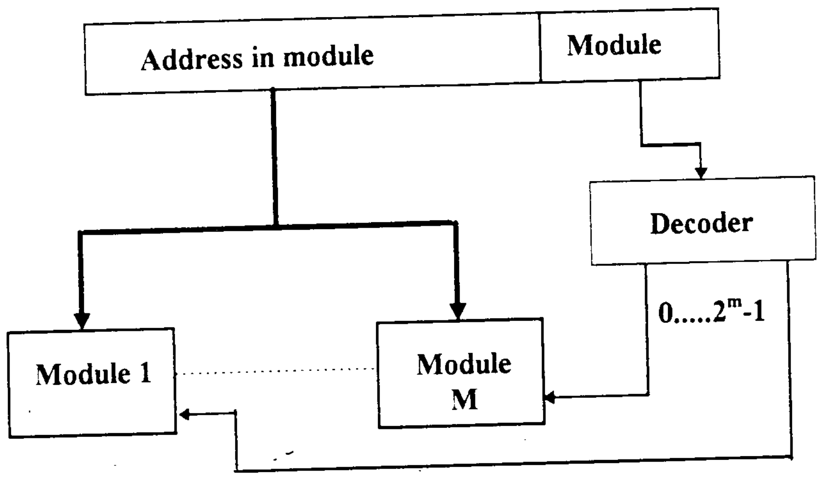


Figure 1.4 : Low order interleaving.

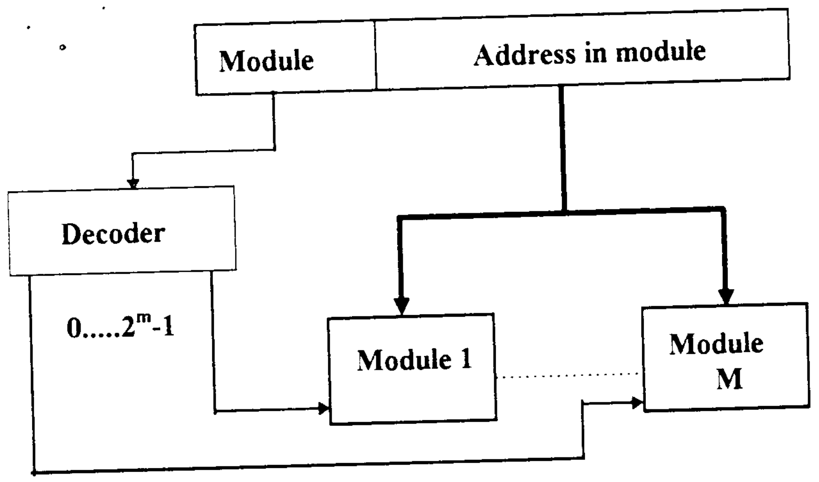


Figure 1.5 : High order interleaving.

Another way is based on parallelism, which in this context implies the use of wider mem y data paths. For example, if the memory is divided into two parts, one part is used to store c and the other to store instructions, the memory bandwidth can be doubled.

1-3.c Execution latency :

Execution latency is the time required for an operation to be executed. The improvement of a computer system at this point is associated with the reduction of the number of clock cycles required for an operation to be accomplished. In the case of simple operation where only one clock cycle is required, the improvement is related to the technological acceleration techniques at the realization level. For complex operations such as multiplication and division, that require longer execution time, the reduction of the number of cycles is a direct improvement of the execution latency. One way used to decrease the number of clock cycles of complex operations is by using specialized arithmetic units requiring less hardware compared to multifunctional units. This results in less propagation delays, yielding an optimized execution latency.

Another approach used to minimize the execution latency is the development of *co-processors*, which are specialized processors used to intervene in the case where complex operations are to be performed. In the most recent computers, a floating-point *co-processor* is designed to be attached to each microprocessor. An example for such floating-point *co-processor* is the INTEL 80287,80387 which can be attached to 80286 [1], 80386 [1] CPUs respectively. In the INTEL most recent microprocessor the floating point *co-processor* is integrated in the 80486 μP.

1-3.d Execution bandwidth:

The execution bandwidth is associated with the number of instructions executed per second (time rate). Thus, to have an idea about the improvement of the execution bandwidth, it is useful to consider the different phases of the microprocessor instruction cycles which are shown in Figure 1.6.

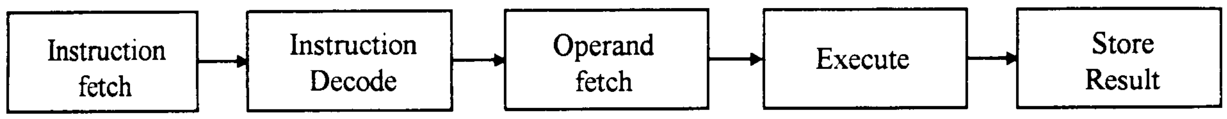


Figure 1.6 : Microprocessor instruction execution cycles.

The CPU instruction execution passes through the following five cycles:

-Instruction fetch (IF): The instruction is fetched from memory and stored in some register.

-Instruction decode (ID): The previously fetched instruction is decoded.

-Operand fetch (OF): The operand address is computed and the data is fetched.

-Execution (E): The operand and the operation to be performed are passed to the execution unit (E), where the result is calculated.

-Store result (S): The result from the operation is stored in an adequate memory location.

If these stages are performed in a sequential manner, only one instruction is executed at a time (Figure 1.7).

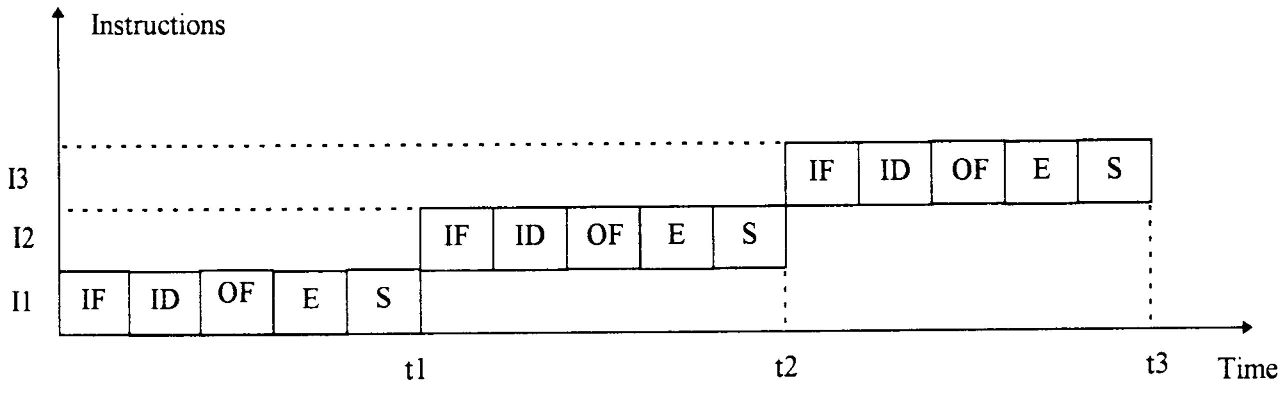


Figure 1.7 : CPU instruction execution cycles in serial manner

One approach used to increase the execution bandwidth is by pipelining the five stages of the instruction execution cycles as it is illustrated in Figure 1.8. With such an approach, the improvement of the execution bandwidth is achieved through the fact that several operations are being processed simultaneously. However, when the five units are designed to handle the CPU instruction execution cycles concurrently, several problems may arise that may cause the five execution units from being overlapped. For example, problems resulting from the pipelining approach applied to INTEL 8086 [27] can be encountered in one of the following conditions: The first condition occurs when an instruction requires access to memory location not in the queue. The second problem which may affect the performance of the processor, occurs when a branching instruction is encountered. The third condition occurs when a complex operation such as multiplication is to be executed. This latter can be avoided by providing more than one execution unit so that regular pipelining may proceed in a regular manner.

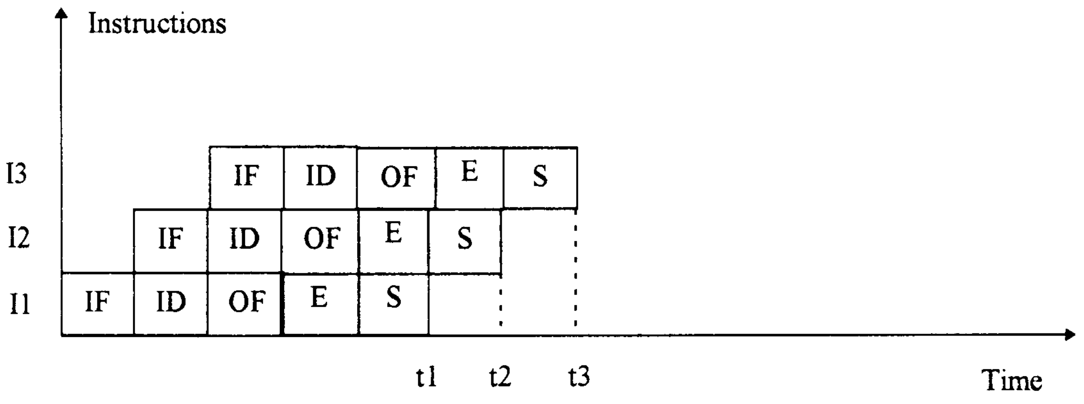


Figure 1.8 : Pipelining the CPU instruction execution cycles.

Most of the acceleration techniques discussed in the previous sections concerning the implementation level have been exploited. Therefore, acceleration mechanisms other than those at the implementation level are to be sought.

1-4 Accelerations at the architectural level:

Due to the limitations encountered at both the realization and implementation levels, computer designer were pushed to look for evolutionary performance improvements at the architectural level.

The acceleration technique investigations at the architectural level set light on three ways to improve computer system performances. The first way is concerned with the design of special-purpose architecture optimized for specific applications. The second technique is oriented toward the reduction of the computer hardware complexity, known as RISC architecture [6,8,23]. The third trend is oriented toward multiprocessor systems. Different types of multiprocessor systems, proposed to achieve higher performance are presented in Chapter 2.

1-4-1 Special purpose architecture:

The optimization of the computer architecture for specific application improves the computer speed compared with that of general purpose computer. Several processors are designed to fulfill the requirements of signal processing, known as digital signal processors (DSP's) such as Fast Fourier Transform, Matrix operations, or Digital Filtering operations in which huge amounts of data are manipulated and complex computation are to be performed in a short time. In general, special purpose processors consist of an ALU, a multiplier, an AND/OR barrel shifter, some address calculators, a program sequencer and a local memory. Special purpose architecture processors are not only used for signal processing, but also for display controllers, data communication, and disk controllers.

1-4-2 Reduction of hardware complexity:

The history of computer growth shows a significant increase in computer instruction size and complexity. By the 1970s, it appeared that much of this complexity that has grown into the computer architecture was there because it was possible, not because it was necessary. Then, it has been realized that it has a great influence on its performance. The complex instructions implementation influence the CPU decoding and execution time. The different types of operands influence the memory organization and addressing modes. This situation compelled a number of investigators to search for an optimum computer architecture.

The objective behind the computer architecture optimization is the simplicity in instruction decoding and execution to achieve an optimal performance. This approach has led to reduced-instruction-set computer (RISC) architecture [6,8,23]. Therefore, a more powerful general purpose computer architecture can be obtained. Its high performances can be achieved from the fact that its design is based only on most frequently used instruction, simple addressing modes, and small number of data types. As the number of instructions is reduced, a smaller number of bits is needed for their representation. Thus, a simple hardware circuitry is required for their implementation. This implies the achievement of a smaller propagation delay.

The main features of RISC architecture that distinguish it from the complex instruction set computer architecture (CISC) are:

- relatively few and simple instructions:

The reduction in the number of instructions allows a small, and simple instruction set format. An 8 bits field can be used to represent the most frequently used operations. The simple instructions permit simple decoding and fast execution. Under this concept, the complex operations need to be interpreted.

- a fixed instruction format.

A large number of instruction format means that a given bit field has different interpretations, depending on the other fields. Consequently, instruction decoding takes a longer time. The use of a fixed instruction format speeds up the decoding operations.

- hard-wired, rather than microcoded, instructions.

The hard-ware implementation of instructions allows faster execution.

- Only a few and simple addressing modes.

The reduction in the deferent addressing modes avoid operand address computations. Simple addressing modes tend to find the operand address in one cycle.

- memory transfers occur only with LOAD/STORE instruction.
- large register set.

As the register responds to the CPU request in less time than the main memory, a large number of register is used to contain the most frequently used operands. This allows a shorter operand fetch time.

Although, some new improvements in the RISC architecture are offered, keeping a computer system as a uniprocessor is not sufficient. A demand for even faster computer still hold on. Thus, to have a significant increase in tomorrow's computer system speed, computers should be able to perform many operations in parallel. This means that a computer should consist of a set of processors that work in parallel to accomplish a certain task. With such approach, the execution of programs can be further accelerated using more than one processor.

2-2 Reasons for architectural classification :

With the advance of technology, there has been a rapid growth in the number of proposed and constructed architectures. As a result of this growth it has been stated that it is not clear which architecture has the best prospects for the future. Then, it has become important for computer system designers to look for taxonomies through which the several models can be easily distinguished. The main reasons of such architectural taxonomies are :

- Architectural model classification permits the computer system designer to know and understand all what has been achieved in the computer architecture field.

- When all the existing systems are classified according to fixed factors, the gaps in the classification can suggest other possibilities which may lead to new improvements.

- The last reason for this classification is that it allows useful models of performance to be built and used.

2-3 Flynn's classification:

Any computer, whether sequential or parallel operates by executing instructions on a given set of data. A stream of instructions (algorithm) tells the computer what to do; at each step a stream of data (input to the algorithm) is processed by these instructions. Depending on the number of streams, Flynn's classification (1966) [3] distinguishes four classes of computer architecture, as shown in Figure 2.1.

	Single data stream	Multiple data stream
Single instruction stream	**SISD**	**SIMD**
Multiple instruction stream	**MISD**	**MIMD**

Figure 2.1 Flynn's classification of computers.

2-3.a SISD (single instruction single data) computers:

A computer of this class consists of a single processing unit receiving a single stream of instructions that operate on a single stream of data as shown in Figure 2.2.

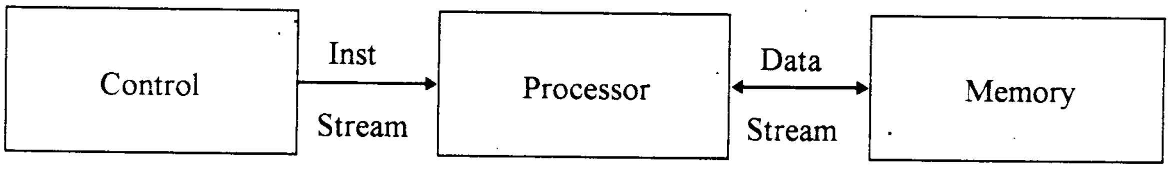


Figure 2.2 : SISD computer system.

All computers, which are based on the von-Neumann architecture and operating in a sequential or serial manner belong to this category.

2-3.b MISD (multiple instruction single data) computers:

N processors, each with its own control unit, share a common memory unit as shown in Figure 2.3. There are N streams of instructions and only one data stream. During the operation of such a system, all the processors execute simultaneously different instructions on the same input data.

This class of computers has a limited number of applications and has been a subject of controversy as the computations require a single input data bank to be processed by several operations, giving several output results.

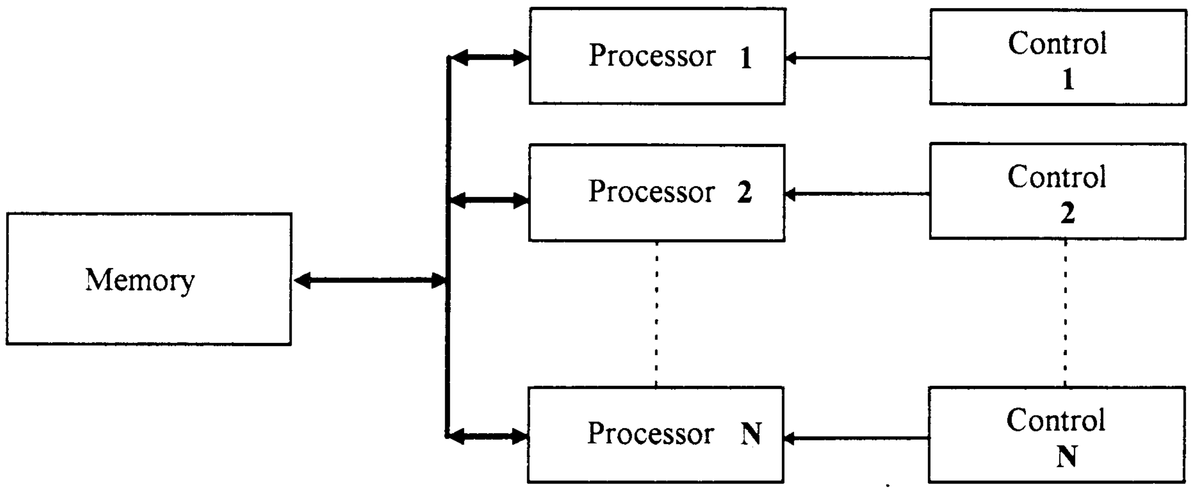


Figure 2.3 : MISD computer system.

2-3.c SIMD (single instruction multiple data) computers:

All the processors operate under the control of a single instruction stream issued by a central control unit. Each processor possesses its own local memory as shown in Figure 2.4.

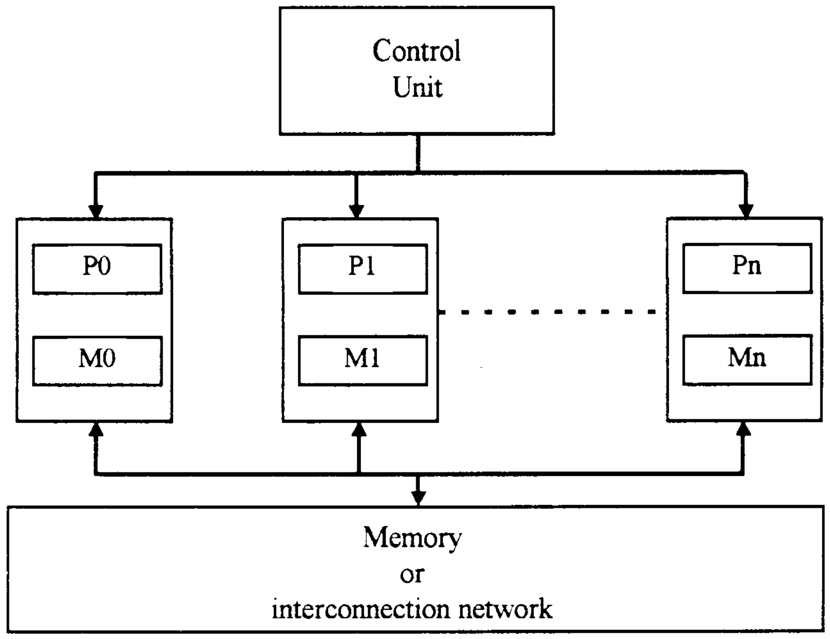


Figure 2.4 : SIMD computer system.

With the use of the SIMD computers, the problem of interprocessor communication arises either with the use of a shared memory or an interconnection network (crossbar switch) . When the communication is established through the shared memory, difficulties will be encountered if several processors attempt to write or read simultaneously data from a given location.

2-3.d MIMD (multiple instruction multiple data) computers:

This class of computers is the most powerful and the most frequently used in parallel system models. It is distinguished from the previous ones by having N processors, N streams of instructions, and N streams of data, as it is shown in Figure 2.5.

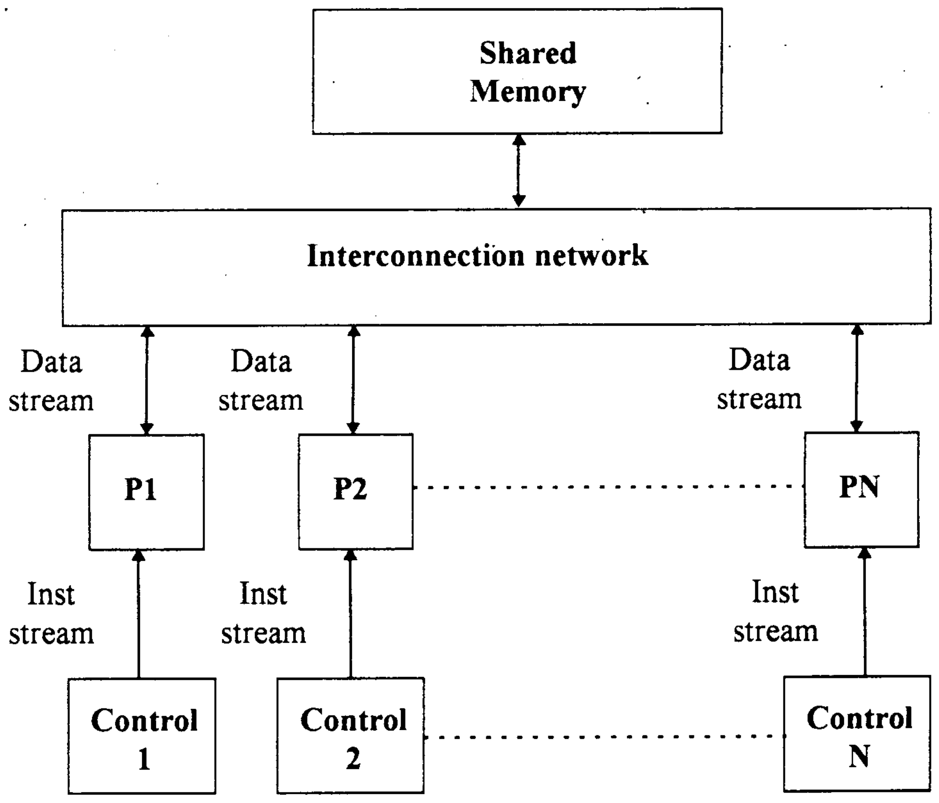


Figure 2.5 : MIMD computer system.

Each processor has its own local control unit and communication between processors can be established through two ways: the shared memory (tightly coupled computer system) in which case communication between processors obeys a certain policy to avoid memory access conflicts, interconnection networks (loosely coupled computer systems) in which case messages are passed between processors according to a specific protocol of communication. This is commonly termed message passing concept.

However, with the rapid growth of the proposed and constructed architectures, many novel constructs did not fit well within Flynn's taxonomy. Two other classifications have been devised and proposed.

2-4 Treleaven's classification:

While Flynn based his classification upon instruction and data streams, Treleaven (1984) [3] has given more importance to control and data mechanisms.

The control mechanisms determine how the computation will proceed. Four mechanisms can be distinguished: control driven, data driven, demand driven, and pattern driven. In the first aspect, the computation proceeds according to a sequence of instructions which are encoded and stored in memory. In the data driven, the key factor driving the execution of the program is the availability of the data. The computation, in the demand driven, proceeds only if the output result is demanded. In the last mechanism, execution takes place only when certain condition are satisfied.

The data mechanisms define the manner in which computational units exchange data between each other. Two possible aspects can be distinguished: shared data and message passing. In the first mechanism, only a single copy of data in memory can be accessed by any computational unit.

In the message passing, each computational unit possesses one copy of the data; whenever a copy needs to be passed from one unit to another, it is copied or passed as a message to the destination unit.

Eight different types of architectures have stemmed from the crossing of these two mechanisms as shown below in Figure 2.6.

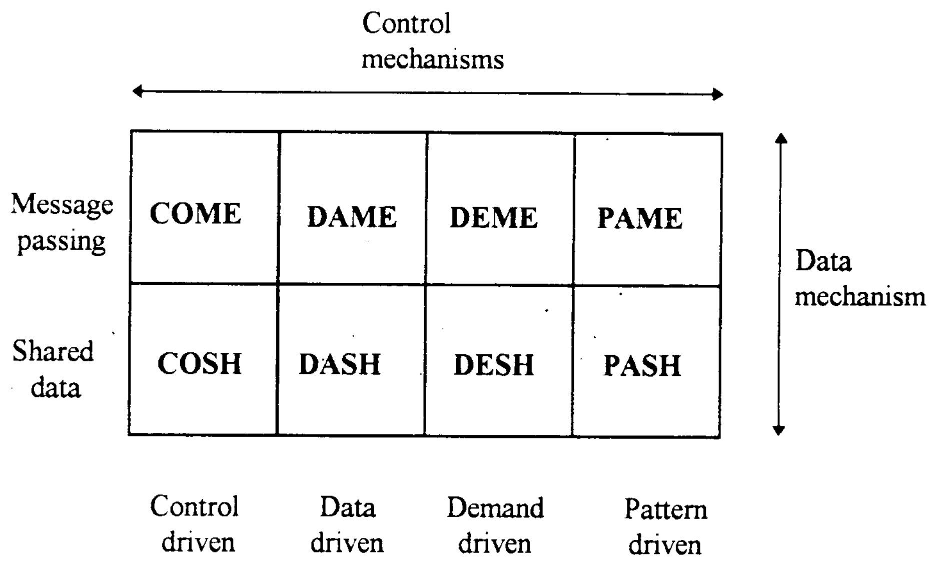


Figure 2.6: Treleaven's computer classification.

2-4.a COSH : (Control driven with Shared data)

The **COSH** combines the feature of a control driven mechanism with the shared data mechanism. The main feature of this type of configuration is a centralized control and a code acting on a shared data. The COSH model is implemented by conventional microprocessors such as the MC680020 computer and conventional computers such as the Vax-11.

2-4.b COME : (Control driven with message passing)

The COME model combines the features of the control driven with the message passing mechanisms. Such a model does not allow the sharing of data. Data between computational units are passed as messages which are copies of data. A pure COME model has not yet been implemented.

2-4.c DASH : (Data driven with shared data)

The DASH computational model assumes the data driven computation combined with the shared data mechanism. A data driven computer involves high degree of parallelism and requires a large number of computational units. However, synchronization problems related to the shared data mechanisms outweigh the parallelism benefits.

2-4.d DAME : (Data driven with message passing)

The computational model DAME assumes that the ignition of programs execution depends on the availability of the data or messages within each computational unit. The natural structure of DAME type computer is parallel. If a set of computational units have their data available, they may proceed concurrently. One of the most known computers based on the DAME computational model is the data flow architecture [17,22].

2-4.e DESH : (Demand driven with shared data)

The DESH computational model combines the demand driven mechanism with shared data mechanisms. The execution ignition starts with an initial quiery for an expression evaluation. The evaluation can proceed as long as all the arguments are available. When an argument is needed, a subquiery for its evaluation is performed. When the argument is evaluated, the computation proceeds with the original quiery. Initial quiery and subquieries are stored in a shared memory.

Consequently, optimal argument evaluation can be obtained. Once an argument is evaluated, it does not need to be computed again. In the DESH model a set of computational model can proceed simultaneously in the evaluation. Like in the case of DASH models, synchronization problems related to the shared data mechanism outweigh the parallelism benefits.

2-4.f DEME : (Demand driven with message passing)

The DEME model is similar to the DESH model except that it assumes demand-driven evaluation with message passing instead of shared data. The message passing mechanism offers to the DEME model the opportunity for parallel evaluation while avoiding the synchronization problems related to the shared data mechanisms. The drawback is the need for the evaluation of the same sub-expression several times, if many computational units require this evaluation.

2-4.g PASH : (Pattern driven with shared data)

In The PASH computational model the execution is driven by goal statement, located in shared memory. Each computational unit is equipped with a pattern matching device. In each computational unit, the ignition of the execution is caused when a pattern matching occurs.

The PASH model offers a good possibility for parallel execution as long as many patterns can be matched in parallel.

2-4.h PAME : (Pattern driven with message passing)

The PAME is similar to the PASH model except that the execution ignition is driven by messages, consisting of data patterns. An initial pattern is used to start the execution. Then , upon the arrival of a message, the pattern contained in the message is matched against the one present in the receiving computational unit.

2-5 Skillicorn's classification :

This taxonomy can be considered as an extension of Flynn's classification. Skillicorn (1988) [9] has based it upon the function structure of the architecture and the data flow between its component parts. The computer system is seen as consisting of four functional units:

- An instruction processor (IP), that interprets the machine instructions.
- A data processor (DP), that acts on data.
- A memory hierarchy , a storage device for both data (DM) and instructions (IM) that passes data to and from the processors.
- A switching network (SW), that insures the connectivity between the other functional units.

According to this functional view, the von-Neumann machine consists of a single IP, a single DP and two memory hierarchies (DM and IM) as shown in Figure 2.7.

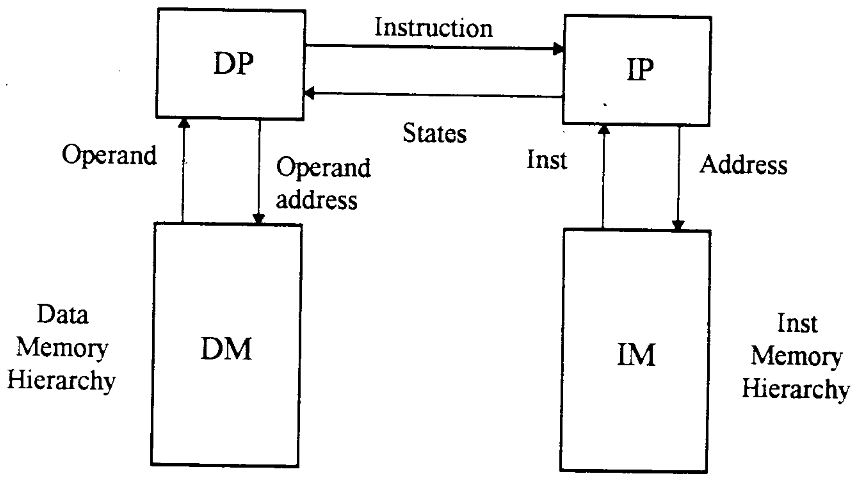


Figure 2.7: von-Neumann machine structure.

The switching network has no role in this class. Based on the model, a large number of classes has resulted by replicating functional units and combining them in different manners. The different possibilities by which the multiple functional units are interconnected leads into several classes of architectures. Four different forms of abstract switches can be used to connect the functional units together:

- 1-to-1 : A single functional unit of one type is connected to a single functional unit of another.

- n-to-n : In this configuration, the ith unit of one set of functional units is connected to the ith of another.

- 1-to-n : In this configuration, one functional unit is connected to all n devices of another set of functional units.

- n-by-n : In this configuration, each device of one set of functional units can communicate with any device in the second set and vice versa.

These architectures can be classified according to the following specifications:

- the number of IP (nIP),
- the number of DP (nDP),
- the number of memory units (nDM, nIM),
- the connectivity between IP's and DP's (IP-DP),
- the connectivity in DP's (DP-DP),
- the connectivity in IP's and IM's (IP-IM),
- the connectivity in DP's and DM's (DP-DM).

On this basis, 28 possible classes of architectures can be determined as it is illustrated in Table2.1.

Table 2.1 Skillicorn's possible architectures

Class	IPs	DPs	IP-DP	IP-IM	DP-DM	DP-DP	Name
1	0	1	none	none	1-1	none	reduct/data uniprocessor
2	0	n	none	none	n-n	none	separate machine
3	0	n	none	none	n-n	n by n	loosely coupled reduct/dataflow
4	0	n	none	none	n by n	none	tightly coupled reduct/dataflow
5	0	n	none	none	n by n	n by n	
6	1	1	1-1	1-1	1-1	none	von Neumann uniprocessor
7	1	n	1-n	1-1	n-n	none	
8	1	n	1-n	1-1	n-n	n by n	Type 1 array processor
9	1	n	1-n	1-1	n by n	none	Type 2 array processor
10	n	n	1-n	1-1	n by n	n by n	
11	n	1	1-n	n-n	1-1	none	
12	n	1	1-n	n by n	1-1	none	
13	n	n	n-n	n-n	n-n	none	separate von Neumann uniprocessors
14	n	n	n-n	n-n	n-n		loosely coupled von Neumann
15	n	n	n-n	n-n	n by n		tightly coupled von Neumann
16	n	n	n-n	n-n	n by n		
17	n	n	n-n	n by n	n-n		
18	n	n	n-n	n by n	n-n		
19	n	n	n-n	n by n	n by n		
20	n	n	n-n	n by n	n by n		
21	n	n	n by n	n-n	n-n		
22	n	n	n by n	n-n	n-n		
23	n	n	n by n	n-n	n by n		
24	n	n	n by n	n-n	n by n		
25	n	n	n by n	n by n	n-n		
26	n	n	n by n	n by n	n-n		
27	n	n	n by n	n by n	n by n		
28	n	n	n by n	n by n	n by n		

CHAPTER 3

PARALLEL SYSTEMS PROGRAMMING

3-1 Introduction:

When the programming aspect is taken a step further, parallel processing systems fall into two broad classes: Shared memory systems and distributed memory systems. This is generated by their different approach to memory organization.

While the first kind of systems is easy to program and has the advantage of using the software designed for sequential machines, the second needs the creation of new programming languages and software supports which are practically inexistant. This fact has not prevented computer designers from giving more importance to distributed systems as they provide more capacities in solving parallel problems.

A parallel program is composed of many processes that have to be executed simultaneously on different processors. These latter need to work together and should be organized in a predefined order to achieve the overall task.

A process scheduling policy is discussed to introduce the method of tackling a parallel program according to the graph theoretical approach [15,47]. Finally, a way of measuring the performance of any parallel program is given.

3-1 Shared memory system programming:

Within such systems , a set of conventional processors are connected through a system bus to a single shared memory (Figure 3.1). Its programming is sequential at each processor level. The interprocessor coordination is accomplished through the global memory. Parallel programming with shared memory systems is easy because of its similarities with operating systems programming and general multiprogramming. However, it is difficult to scale up to a large number of processors. This drawback is due to the saturation of the bus when the shared memory is demanded by many processors at the same time.

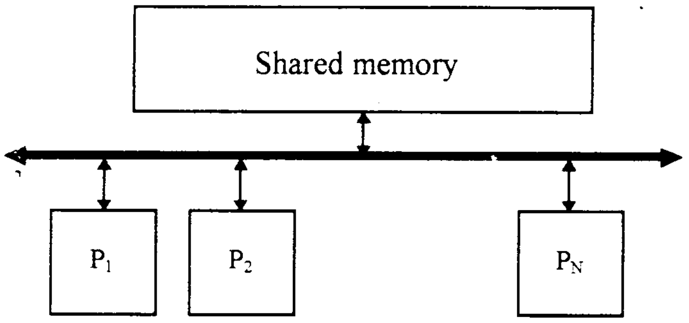


Fig 3.1: Shared memory system.

Several architectural solutions have been proposed to remedy this shortcoming. One solution is to use a master processor to grant access to the bus using time sharing. Still there remains a serious memory bandwidth problem: if P processors want to access the memory, this will take P times longer than a single processor. This was partially solved by using the memory interleaving method; the memory is divided into several independent modules that are connected to the processors through a crossbar switch (Figure 3.2).

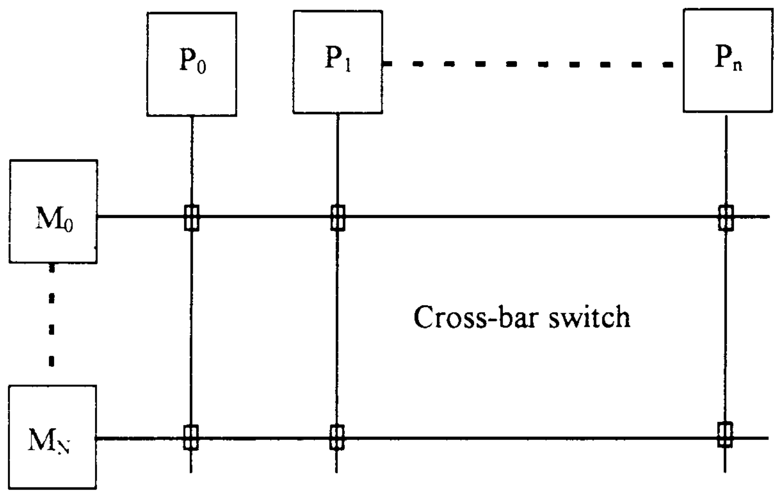


Fig 3.2: Memory interleaving approach

With such a configuration, only one processor can access one memory module at a time. But they can access many modules at a time.

A third solution is the use of cache memories that have copies of the global shared memory as shown in Figure 3.3.

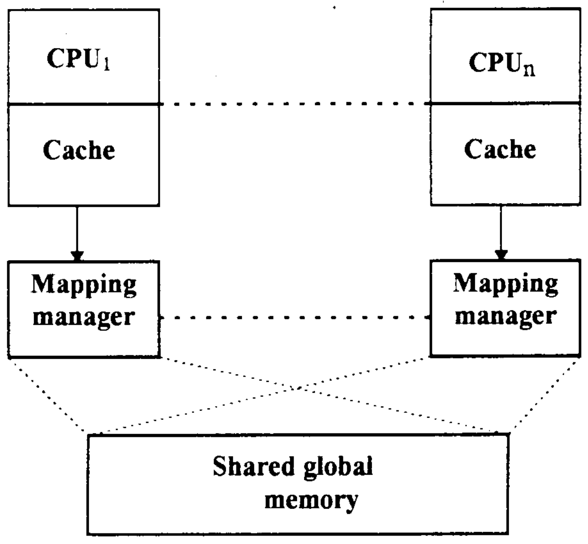


Figure 3.3 : Shared global memory mapping

In such a configuration, bus saturation can be reduced as long as all the instructions and data ıre to be fetched and loaded in the local cache memories. However, this technique may lead to the ɜmory coherency problem [31]. The cache memories are said to be coherent if all the data values ..re the same. A memory manager and a cache coherency algorithm [37] are needed to manage the ıapping between local cache memories and the global memory. It can be inferred that this t :chnique adds complexity to the initial problem.

A recent method that has been made possible by technological improvements at the realization ıevel is the use of Multiport parallel random access memory (PRAM) [36] (Figure 3.4.).The number of ports being less than 4 and the size of the memory not exceeding the 16 Kbytes barrier ninder a wide use of these memories and promise a future expansion.

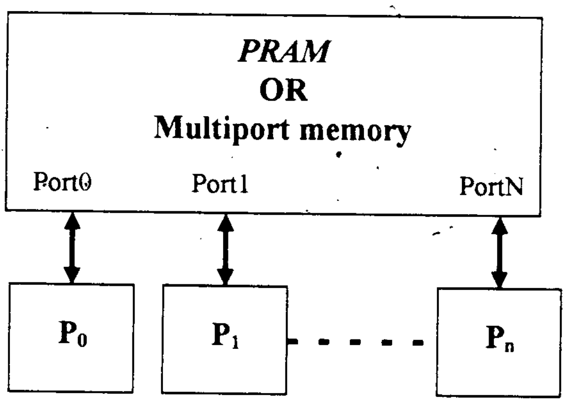


Figure 3.4 : PRAM system

3-2 Distributed memory system programming:

As it can be noticed in the last section, all the solutions that have been proposed tend to allocate to each processor its own local memory. This fact has naturally led to the concept of distributed memory systems, where every unit is a stand-alone processor with its proper memory.

The interprocessor communication is established by message passing. Distributed systems are often called message passing systems. There are two ways to achieve the transfer of data between processors: synchronously or asynchrounsly. In the first, the two processors involved in the communication are fixed at a rendez-vous point. In other words, when one is still executing the other is blocked or delayed until both of them are ready. For the second method, the sender processor passes the message regardless if the receiver processor is ready or not. This calls for a buffering operation that has to be supported by an interface between the two processors.

Distributed memory systems are given the advantage over the shared memory systems by the fact of being scalable (i.e. the communication bandwidth and the overall system performance are maintained when a large number of processors is used).

There are, however, two major problems associated with the programming of distributed systems that affect the overall system performance : communication time delays and load unbalance. If the cost of communication during the execution of a parallel program is too high, the processors spend most of their time transferring data through the system interconnection network. For the second issue, the system is unbalanced which means that the tasks are not well allocated resulting in processors that execute many tasks while others are sitting idle. The system resources are not used to their limits. Therefore, the design of parallel programs on distributed systems requires a mapping from the sequential space to the parallel space. This mapping is achieved through process scheduling. This feature is so important to parallel programming that the following section is devoted to its description.

3-4 Task allocation scheduling:

In the design of parallel programs on distributed systems, a set of processors are supposed to be connected in such a way that a parallel target machine is constructed. The scheduling requires that the program be decomposed into elementary tasks. These tasks have to be assigned to the processors according to a certain policy depending on the tasks interdependence and the limitation of the target machine.

The scheduling technique can be either local or global [47]. The first one deals with concurrent processes in single processor systems. The global scheduling handles the allocation of tasks in multiprocessor systems. The global scheduling can be performed either in a dynamic [14] or in a static [30,43] manner. In the static task scheduling, the information regarding the tasks such as execution time of each task, intertask or interprocessor communication time, task execution precedence relations or their execution order is known prior to the task allocation. It is called static or deterministic task allocation scheduling techniques. In the case of dynamic task assignment and scheduling, no prior fixed information about the tasks is given, and the number of tasks or system resources could change during the system runtime. Task assignment and scheduling decisions must be made during system runtime in order to satisfy the changing requirements. Static task assignment and scheduling techniques have received a lot of attention in parallel program design. As the design of the Multiple Scale Signal Matching (MSSM) algorithm of interest in this thesis is based on the static task allocation model known as graph theoretical method, we survey the graph theoretical approach here.

3-4.a Graph theoretical method:

There are three main aspects that have to be considered to build a graph theoretical system. The first two constituents are the *task graph* and *the target machine graph*. In the task graph the tasks are assigned numbers and are represented by nodes. Each node is split into two halves, the upper half indicates the task number and the lower one gives the execution time of the tasks. The data dependency and execution order are represented by directed arrows that are labeled with the communication time delay (Figure 3.5).

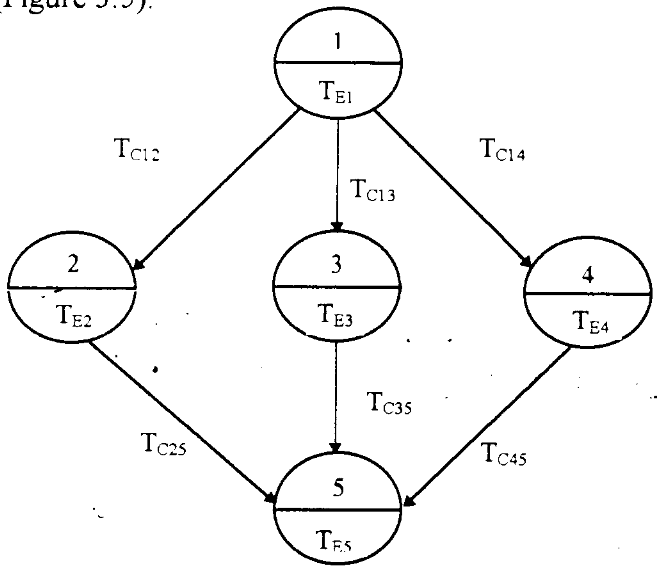


Fig 3.5 : Program task graph

In the target machine graph the processors are given numbers and are represented by nodes. They are linked by directed arrows labeled to indicate the interprocessor connection (Figure 3.6)

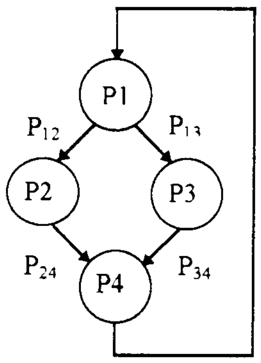


Fig 3.6 : Target machine graph.

The remaining stage is the scheduler that takes care of the decision of allocating tasks to resources according to some policy. The combination of the task graph with the target machine through the scheduler gives out the *Gantt Chart*. It represents the different processors on one axis and the task execution time on the other one (Figure 3.7). After the task assignment, the Gantt Chart can serve as a means of system performance measure and will eventually give an idea about the resource (processors) utilization or the system load balancing.

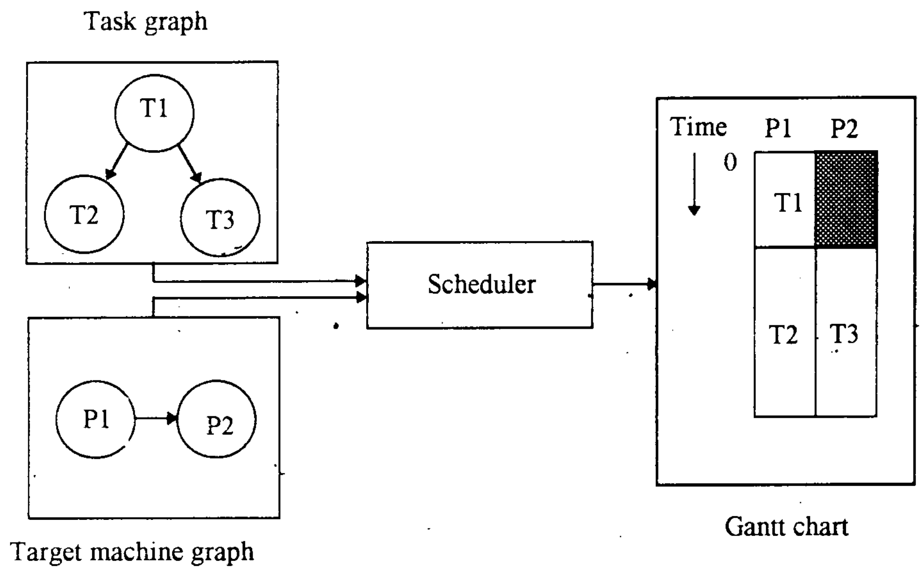


Fig 3.7 : Graph theoretical system approach.

The task allocation scheduling based on the *graph theoretical approach* requires that the algorithm under development be partitioned into a set of tasks to set light on the inherent parallelism of the algorithm. The purpose of task allocation scheduling on a set of interco nnection processors is to reduce the job turnaround. Such an objective can be achieved by maxim ng the resource utilization, while reducing the time that can be spent in communication. It is a naxmin problem because a trade off between system load balancing and minimizing the rhead communication time is encountered. While trying to minimize the interprocessor communication time delay, the tasks tend to be assigned to a single processor on which they are executed in sequential manner. In such a case, the system load balancing is lost. On the other side, if the tasks are assigned in such a way that a best load balancing is obtained, more overhead communication delay is added to the overall execution time. Under such constraints, if the number of the processing elements is increased, a higher execution time may result even if the system is well balanced. The increase in the execution processing time can be explained by the increase of the interprocessor communication time, which is due to the precedence order of parallel tasks that are assigned to separate processors.

Based on these two aspects, the allocation strategy tends to assign the heavily attached tasks on the same processor. In other words, if the communication time delay between two attached tasks is very large, a shorter execution time can be obtained if both of them are allocated on a single processor. To illustrate better this idea, let us take a look at the Gantt Chart shown in Figure 3.8.

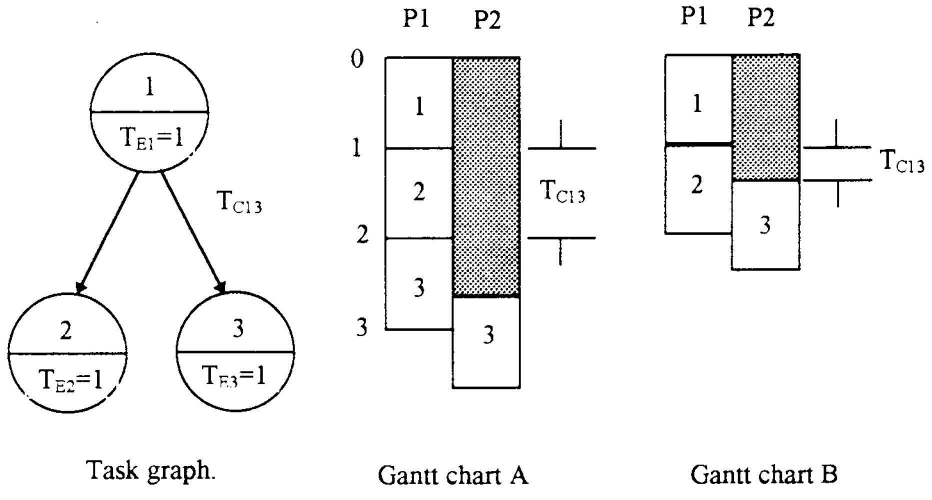


Figure 3.8 : The task allocation consideration due to communication time delay.

It can be noticed from Gantt Chart A that if the communication time T_{C13} between task 1, and task 3 is greater than the execution time of task 2, the starting time of task 3 on P2 is later than its starting time on P1. When all the tasks (1, 2, and 3) are mapped on processor P1, the execution time (T_E) of this task graph is the summation of the execution times of tasks 1,2, and 3 as described by the following equation:

$$T_E = T_{E1} + T_{E2} + T_{E3} \qquad \text{Eq 3.1}$$

When task 3 is assigned to processor 2, the execution time of this task graph is the summation of task 1, 2, plus the communication delay T_{C13}. This can be described by the following equation.

$$T_E = T_{E1} + T_{E2} + T_{C13} \qquad \text{Eq 3.2}$$

It can be noticed from equation 3.1 that when all of the tasks are allocated on a single processor (P1), the execution time is equal to three units of time. From equation 3.2, it is clear that if $T_{C13} > T_{E2}$, the task graph execution time will be greater than if all the tasks are executed on a single processor P1 (more than three time units). Therefore, when $T_{C13} < T_{E2}$, the execution time is shorter than the sequential execution as shown in Gantt Chart B.

Another approach, known as the task duplication approach [27], may be used to offset the communication delay during the task allocation process. The duplication may solve the maxmin problem by duplicating the tasks that influence the communication time delay. As shown in Figure 3.9, T1 is duplicated to run on both P1 and P2. Thus, task T3 can start sooner than if both tasks T2 and task T3 were assigned to the same processor. Therefore, duplicating task T1 gives the advantage of the inherent parallelism of the application and reduces the communication time delay at the same time.

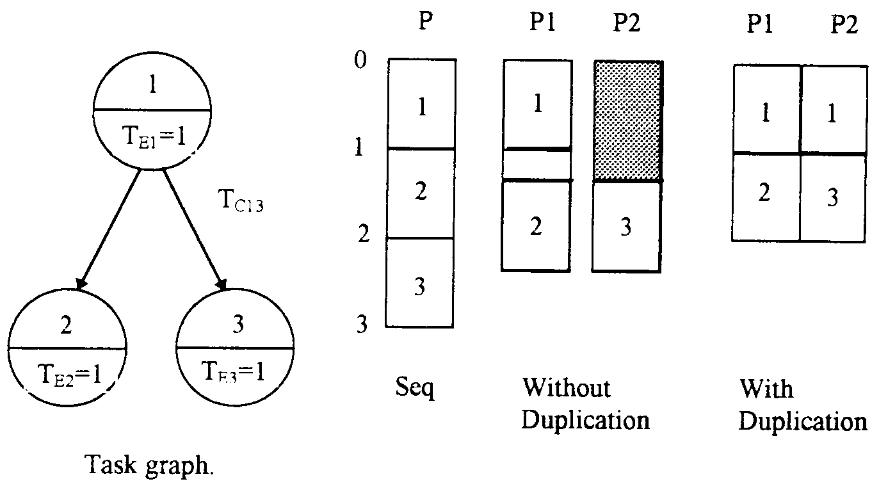


Figure 3.9 : Task duplication approach.

Based on the two approaches described above, it can be noticed that a shortest execution time can be achieved not only by distributing the eventual parallel tasks on as many processors as possible. When parallel tasks are distributed on all possible available processors, the overall communication time increases. This later may affect drastically the overall execution time of the program. Therefore, with the task allocation scheduling, a shortest execution time is obtained by minimizing the communication time delay.

3-5 Performance measure:

The main objective behind the parallel design of algorithms is the reduction of their execution time when implemented on a multiprocessor system with respect to their execution time on single processor systems. However, the performance measure of the parallel implementation can be performed through the execution time gain which is defined as the sequential execution time over the parallel one. The time gain [32] is denoted by G, and is represented by the following equation:

$$G = \frac{T_S}{T_P} \qquad \text{Eq 3.3}$$

Where : T_S is the sequential execution time on a single processor system.

T_P is the parallel execution time on a multiprocessor system..

Another factor that can be used for the performance measure of a parallel implementation is the efficiency of the system on which the parallel algorithm is implemented. It is defined as the time gain over the number of processors (transputers) in the network. The efficiency of the system can be represented by the following equation:

$$E = \frac{G}{P} \qquad \text{Eq 3.4}$$

Where P is the number of processors in the multiprocessor system.

By inspecting the two performance measure equations given above, it can be noticed that, if the algorithm is completely parallelized, a maximum efficiency of 1 is obtained. Thus, if the algorithm is completely parallelized, the sequential execution time over the parallel one equals the number of processors in the system as given below:

$$Max(G) = \frac{T_S}{T_P} = P \qquad \text{Eq 3.5}$$

However, in real applications, due to the time spent in communication between processors, the precedence relationship that may exist between parallel algorithm tasks and load balancing issues, the efficiency of the system is degraded. This means that the three above constraints increase the difficulty of arriving to the maximum time gain. Thus, in the parallel design of a parallel algorithm, one is required to overcome these problems, so that an optimal time gain is obtained.

CHAPTER 4

TRANSPUTERS

4-1 Introduction:

During the past ten years, many shared parallel computers have been designed such as Cray-1 [45] and ILLIAC V [11,39] and many distributed memory systems have been designed such as the Butterfly [37] and MPP (Massively Parallel Processor) [29]. In shared memory systems a set of conventional processors are gathered around a shared memory. Such a configuration leads to the bottleneck problem as the interconnected processors share a common bus.

Computer scientists have discovered that distributed memory systems can scale up easily than shared memory systems and lead to the design of massively parallel processors such as the MPP (Massively Parallel Processor). But since distributed memory systems are based on communication networks, the interface between the processors and the communication networks is a burden. It was complex and slow on one hand and on another hand, the CSP (Communicative Sequential Processes) model [5] could not be implemented at the language level.

Recently, with the advance of the integration techniques to very large scale (VLSI), very powerful interconnected processors have been designed to construct multiprocessor systems to fulfill the requirements of increased computational power and to overcome the bottleneck problem encountered in shared memory systems. The most famous ones are the INTEL/Carnegie Mellon Warp[16], and the INMOS transputer [7,10,25,40].

As the transputer has been used in the implementation of the present project, this chapter is devoted to an insight of its hardware and software aspects, along with its development system.

4-2 The transputer architecture:

All transputers include a processor (16 or 32 bits), a system service, two or more serial links with their link interface, a timer and what makes them members of the distributed memory system family, an on-chip memory that can be expanded through a memory interface (Figure 4.1).

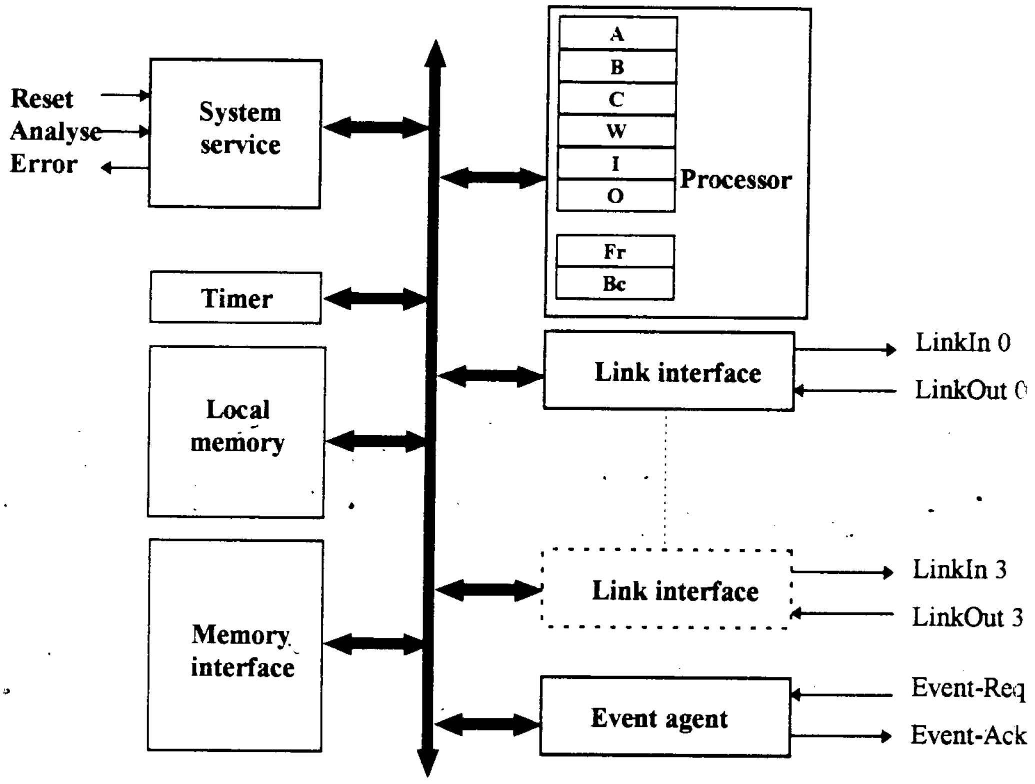


Figure 4.1 Transputer generic architecture

The processor has six registers for the execution of sequential programs: Areg, Breg, and Creg. They are used to evaluate expressions and hold instruction operands and results; the Ireg, a program counter pointing to the next instruction to be executed; the Wreg, work space register that points to the block of local data of the current process; and finally the operand register Oreg. The transputer's processor consists of two other registers that support concurrent processing and are known as the *Front* and *Back* registers. They are used to hold the addresses of the first and the last processes to be executed concurrently.

The system service manages three signals that are important to transputer based systems: R*eset*, A*nalyze* (in the case of tracing), and E*rror* that signals an error occurring during the execution of a program.

The transputer has a pair of event pins that are used to signal to the transputer that some external events have occurred. Events are handled by an on-chip event agent. They are programmed exactly like a link via the Event-req input. When external events occur, the transputer can acknowledge the external devices using Event-Ack output pin.

The transputer has two timers which can be utilized by the programmer for real-time programming, timing events, and delays.

The transputer memory addresses are signed binary representation. This fact reduces the address computation time in the processor's ALU. The address space is organized as follows: The nine bottom words are used by the links and events, the next two upper words are used by the timers. The remaining space is allocated for the interrupt save area, the on chip memory, the external memory, peripherals, and ROM (Figure 4.2).

The transputer links are fundamental to both the hardware and software aspects of the transputer communication concepts. Under this latter, the communication is established through their physical bi-directional serial links using the message passing concept. The transfer of data is orchestrated at a rate of 10 Mbits per second. This speed is totally independent of the processor's computing speed, allowing connection of different transputer versions in the same system.

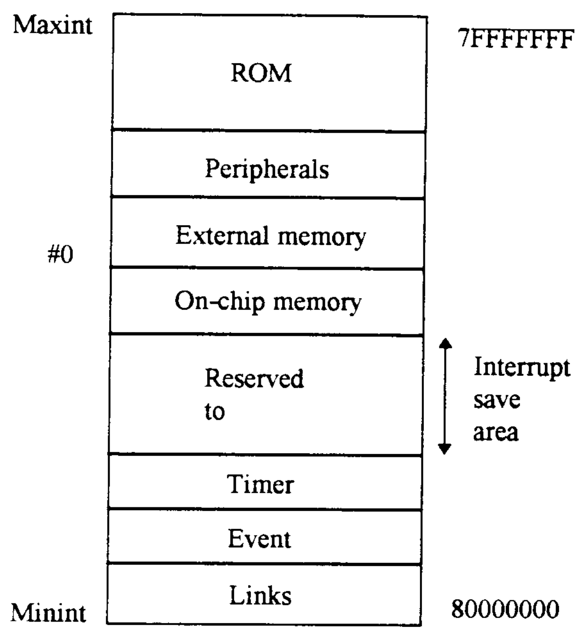


Figure 4.2 : Memory map.

Communication through the links involves a simple protocol that ensures the transmission of a byte at a time in a packet. Each packet consists of two one's, the byte to be sent, and terminated by a zero "0"; (Figure 4.3). The reception is acknowledged by the receiving transputer using an acknowledgment packet as soon as the data packet is identified.

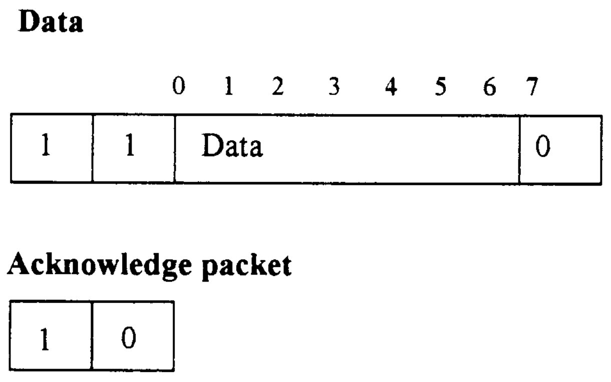


Figure 4.3: Transputer communication protocol

4-3 Transputer instruction set:

All the instructions are one-byte long. The four most significant bits constitute the function field, the other bits are used to represent the operand value (Figure 4.4).

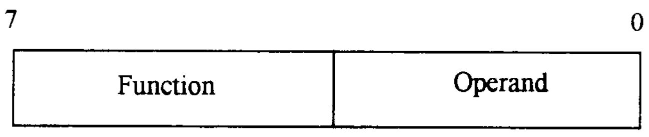


Figure 4.4: Instruction format

This format of instruction is based on the concept of the RISC architecture [6,8,23] that requires a simple decode mechanism, reduces chip area, and increases execution speed. There are three forms of instructions: Direct, indirect, and prefixed.

Of the sixteen possible instructions, thirteen are used to represent the most frequently used functions. They are known as one address or direct instructions as the operand is used by the instruction as a value:

1- Load local
2- Store local
3- Load local pointer
4- Store non-local
5- Load constant
6- Add constant
7- Add to memory
8- Conditional jump
9- Load non-local
10- Load non-local pointer
11- Adjust workspace
12- Jump
13-Call

Another type of instructions use the operand to define operations on values already in the evaluation stack. They are called indirect or zero address instructions. The remaining two instructions are the prefix and Negative prefix that are used to extend the operand of any instruction to the length of the operand register. The prefix instruction starts by loading its four data bits into the operand register to four places, then the data field of the next load instruction is ORed with the four least significant bits of the operand register (Figure 4.5). Consequently, data can be extended to the length of the operand register by a sequence of prefixing instructions. The negative prefix instruction is similar to prefix one except that it performs a 2's complement of the operand register before shifting it.

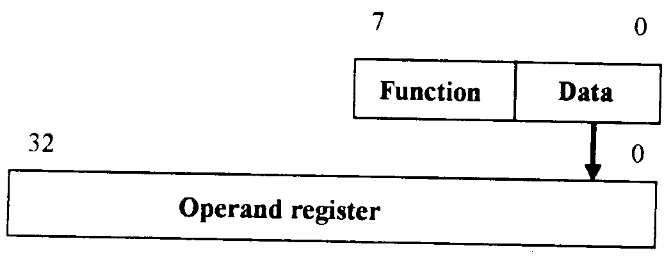


Figure 4.5 : Loading the operand register.

4-4 The OCCAM language:

OCCAM's model of programming is based on CSP (Communicative Sequential Processes) model [40]. It was developed by INMOS to allow a program to be considered as it consists of a collection of concurrent processes that communicate with each other and peripheral devices through channels. Although OCCAM language [4,18,28,41] is an abstract programming language, its development has been closely associated with that of the INMOS transputer [7,10,25,40]. Thus, OCCAM can be considered as the transputer assembly language which provides to the programmer the same programming techniques on a single transputer and a network of transputers. This enables a programmer to be essentially unconcerned about the final implementation scheme.

OCCAM programs are constructed of hierarchical levels of processes. At the lowest level, the language is built up from just three primitive process types: assignment, input, and output. These two latter support interprocess communication via channels as it is illustrated in Figure 4.6.

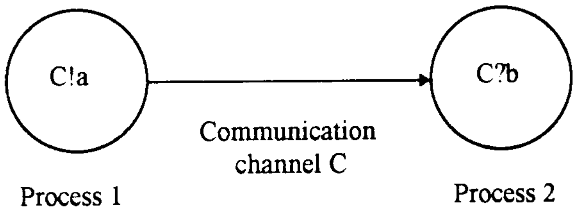


Figure 4.6: process communication

The symbols used for these three primitive processes are:

1) Assignment: ***a:= b*** Assigns expression ***b*** to variable ***a***.

2) Transmit.: ***C ! b*** Outputs the value contained in ***b*** to a previously declared channel *C*.

3) Receive : ***C ? b*** Inputs a value from a previously declared channel *C* and assigns it to the variable ***b***.

OCCAM supports a number of basic variable types: CHAN, TIMER, BOOL, BYTE and INT. OCCAM's channels as well as the type of data to be transmitted or received must be declared in the outer scope of a program as it is illustrated in the following example.

Channels declarations:

```
CHAN OF INT chan1, chan2.
Program
•
•
•
```

This states that both *chan1* and *chan2* are declared as being channels, through which integers are transmitted and received.

OCCAM supports also the signed integer INT16, INT32, INT64, and the floating point type REAL 32, and REAL 64.

Expressions may be constructed using the following operators:

Arithmetic operator: +, -, *, /, \ (\ the remainder)

Modulo arithmetic

Relational : :=, <>,>,<,<=,>=

Boolean operators: AND, OR, NOT

Bit operations: BITAND, BITOR, <> (exclusion OR), BITNOT

Shift operators: <<,>>

Primitive processes may be combined to build up level processes by the use of constructs. The sequential construct, SEQ, indicates the processes that follow it are to be executed sequentially.

Sequential execution of processes:

```
CHAN OF INT Chan1,Chan2:
INT X:
SEQ
   Chan1? X
   X:=X+1
   Chan2! X
```

In addition to this sequential construct, OCCAM provides a construct which enables the parallel execution of processes, thus facilitating interprocess communication. The parallel construct, PAR, indicates that the processes which follow it are to be executed in parallel.

Parallel execution of processes:

```
CHAN OF INT Chan1,Chan2:
 PAR
    ... process 1
   INT x            (x here is local to Process 1)
   Chan1 ? x
    ... Process 2
   INT x            (x here is local to Process 2)
   Chan2 ! x
```

As in many conventional languages, OCCAM's conditional statement is provided through the IF construct.

Conditional statement:

```
CHAN OF INT Chan1,Chan2,Chan3:
INT x,y:
IF
 x=1
   Chan1! y     Output y on channel chan1
 x=2
   Chan2!y      Output y on channel chan2
 ELSE
   Chan3!y      Output y on channel chan3
```

In addition, there is another means of choosing or selecting the process to be executed, which depends on greater number of variables: the CASE construct.

Selection statement:

```
CHAN OF INT Chan1,Chan2:
INT x,y:
CASE X
  1,2,3
    Chan1 ! y     Output y on channel chan1
  4
    Chan2!y       Output y on channel chan2
```

The value of y will be either output on chan1 (if x=1, or x=2, or x=3) or chan2 (if x=4).

With the OCCAM language, choices can also be made according to the state of the channels. Such feature can be accomplished by the ALT construct. The ALT construct can be particularly useful where input values might appear on one of a number of alternative channels.

Alternation statement:

```
CHAN OF INT Chan1,Chan2,Chan3:
INT X:
ALT
  Chan1
     Process 1
  Chan2
     Process 2
  Chan3
     Process 3
```

If one among the three channels produces an input, only the associated process will be executed.

OCCAM offers two constructs that perform repetition. The first one is used to repeat a process for a specified number of times. Such a repetition can be used with the SEQ construct to create conventional loops, and with the PAR construct to build up arrays of concurrent processes, as shown in the following example.

Repetition using SEQ construct:

```
1- SEQ  i=0  FOR N
       PROCESS
2- PAR i=0   FOR N
       PROCESS i
```

The second repetitive construct can be performed by the WHILE construct which includes a test in its execution. The process is executed as long as the test is true.

LOOPS using WHILE statement:

```
INT X,n:
  SEQ
      X:=n
      WHILE X>0
          SEQ
              y:=X+1
              X:=X-1
```

In order to increase the readability and maintainability of the programs, processes can be named and represented by procedures. The procedure is formed by the keyword PROC, the name of the process, and the process itself or the procedure body. This latter is executed whenever its name is encountered during the execution of the program. Procedures may contain parameters which allow different values to be passed to another procedure or channels to be used at its different instances.

Procedure:

```
PROC task (CHAN OF INT C1,C2,C3)

INT X:
ALT
  C1 ? X
     Process 1
  C2 ? X
     Process 2
  C3 ? X
     Process 3
:
CHAN OF INT Chan1,Chan2,Chan3:
SEQ
    task  (chan1,chan2,chan3)
```

4-5 Transputer development system:

The transputer development system (*TDS3*) [19] is an integrated development system that is used to develop OCCAM applications for transputer based networks. It runs on transputer board plagued on an IBM PC, as the INMOS SPRINT board [33] or the B008 board [20]. All the development software utilities such as the Folding Editor, Compiler, and Debugger are integrated in the TDS3 and run on the transputer boards (Figure 4.7).

The interface between the TDS3 environment and the MSDOS or the resources of the IBM PC in general is performed by a program which runs on IBM PC called a SERVER. This latter provides the development system with the access to the terminal and filling system of the IBM PC. The developed program applications can be written, compiled, and loaded on the transputer within the transputer development system.

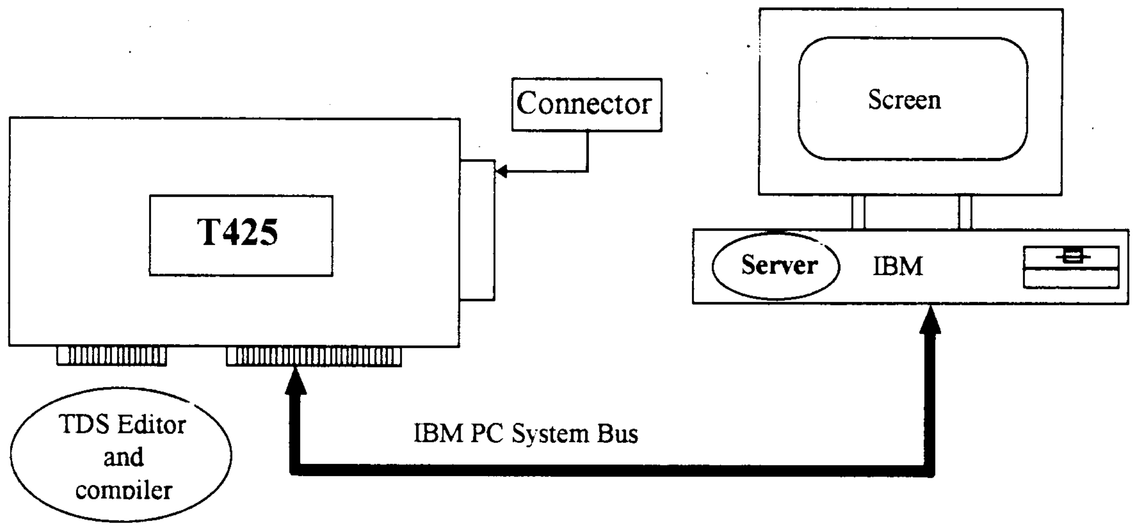


Figure 4.7 : Transputer development system.

IV -5. a SPRINT board description:

The SPRINT board [33] is a half length PC plug-in board. It includes an INMOS T425 [21] transputer, and 2 Mbytes of Dynamic memory. Its PC bus interface is completely *B004* compatible. This latter owes its name from the fact that the first ever transputer board that has been used in a PC machine had the part number *B004*. Since then all the other board manufacturers have produced compatible interfaces, so that the software interface to the transputer remains compatible.

A "*DB25*" socket is provided at the rear of the SPRINT board. All the transputer links, subsystem control signals, the PC link and the PC-controlled system services are taken out of the "*DB25*" socket as shown in Figure 4.8. This allows to the SPRINT board to be connected to other INMOS boards or controlled by an external network.

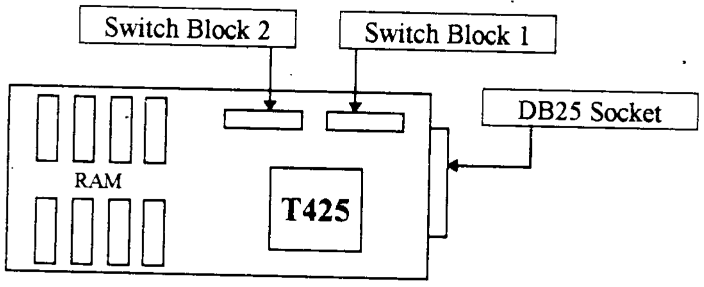


Figure 4.8 : INMOS sprint board.

The SPRINT board contains two set of switches by which several of the operating parameters of the board can be changed. The first block of switches is concerned with the setting of the transputer links speed (from 10 Mbit/sec to 20 Mbit/sec), the transputer clock speed (17.5, 20, 22.5, 25, 30, 35 MHz), and also the transputer memory speed. The second block of switches controls the direction of the link signals from the PC host and the system service of the transputer

(Reset, Analyse, and Error signals). By setting these switches, the service signals can be either connected to the line of the host interface or to the "DB25" socket, allowing the board to be configured, so that it can be controlled from the host PC or an external PC.

The communication between the SPRINT board and the host IBM PC is assured by the CO12 link interface adapter [21] (Figure 4.9). The CO12 device is a chip produced by INMOS used for the conversion of the parallel data from the PC host to the serial data stream that is acceptable by the transputer.

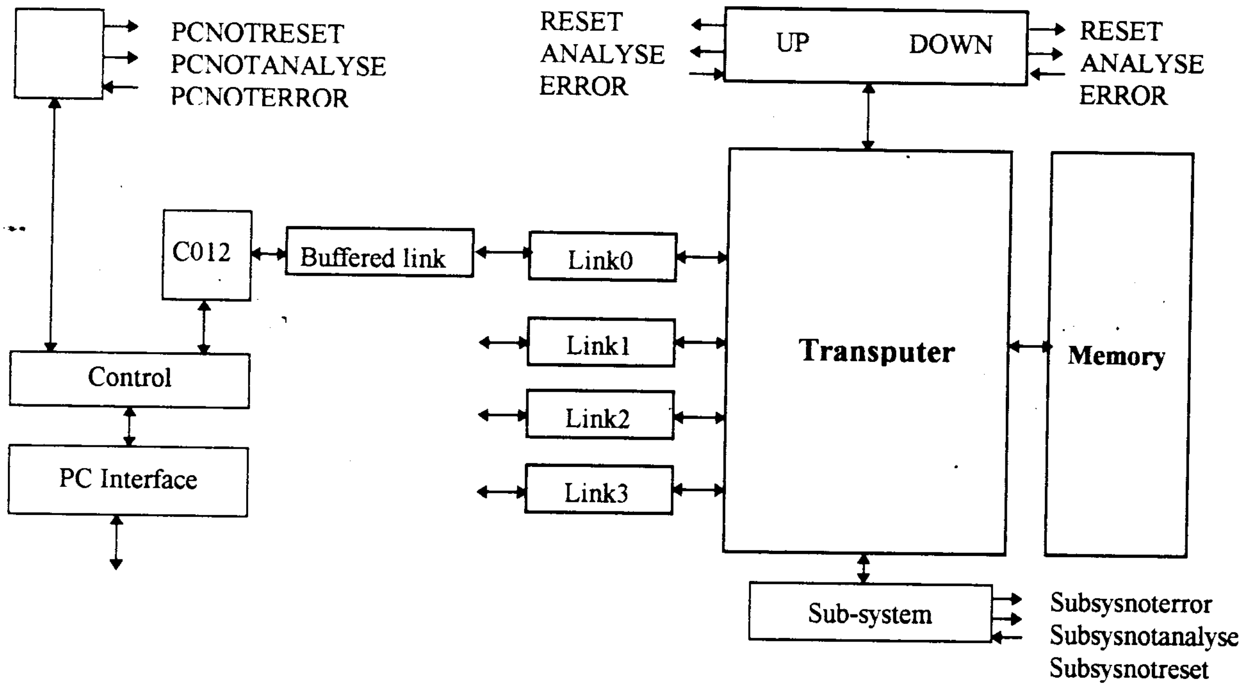


Figure 4.9 : SPRINT board block diagram

4-5.b SPRINT board configuration :

The SPRINT board [33] can be configured in three different manner. The DB25 socket and switch blocks described in the previous sections are provided for this purpose. Depending on whether the on board transputer is to be controlled by host or by another external network as in the case when building a multiprocessor system. The three possibilities are discussed in the following.

1) Transputer controlled from an IBM PC:

This is called a stand alone configuration. In this mode, the SPRINT board is completely controlled by the host. The Reset, Analyse, and Error signals of the transputer are connected to the PC bus interface. In the case when, the ON board transputer is to be used to control an external network (other transputers), the Sub Sys-port or the DownNot-port on the *DB25* socket can be used.

2)- Transputer controlled from an external network:

In this mode, the Reset, Analyse, and Error signals of the transputer are connected to the Up-Not line on the *DB25* socket. This configuration is used when a transputer network is to be built. With this configuration the onboard transputer is completely controlled by an external network.

3)- Controlling transputers other than the one on the board:

In this mode, the link input, and output lines of the CO12 link adapter are connected to the *DB25* socket, and with the three transputer control signals (PCNotReset, Analyse, and Error), the host IBM PC on which the SPRINT board is installed can be used to control other transputers than the onboard one. This configuration is almost unused.

In the transputer based network used to carry out the parallel implementation of the Multiple Scale Signal Matching algorithm [26,38], four SPRINT boards were used. Each one has been inserted on different PC's (IBM compatible) as depicted in Figure 4.10. The Host board has been configured in a stand alone configuration mode (controlled by the Host IBM PC). The remaining boards were configured using the second mode by which they can be controlled by the Host transputer board. More details about the transputer interconnection links will be presented in Chapter 7.

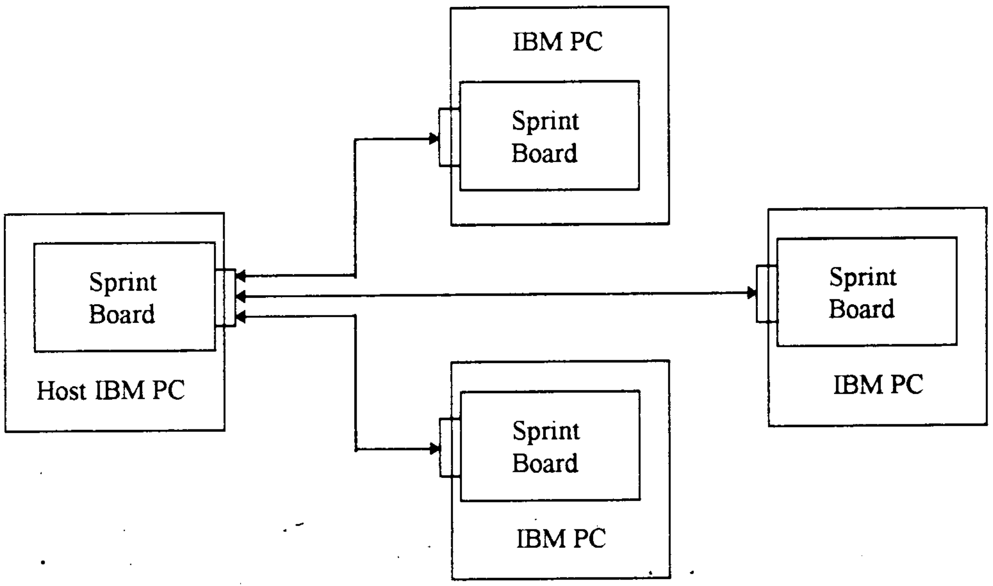


Figure 4.10 : Transputer based network using the INMOS SPRINT board

4-6 Transputer family:

In the first transputer generations, there are three main types: the 32 bit transputers with a floating point unit known as the T8 or T8xx, the 32 bit transputer without floating point unit known as T4 or T4xx, and the 16 bit transputers known as the T2 or T2xx. The first production was the IMS T414 (1985) [21,24]. It has 2 Kbytes of on chip memory and four slower links. The T414 was enhanced into new version termed the IMS T425 [21], which has a 4 Kbytes of on chip memory and fastest links, plus some additional instructions and pins for extra functions.

The T400 [21] is a low cost version of the T425 with only two links and 2 Kbytes of memory. It is available in plastic packaging. The T426 [21] is similar to the T425 except that it includes a memory interface (emi) [25] by which memory capacity can be increased with a minimum of external components.

The second production of the transputer first generation was the 16 bit transputer version. The first version of this category was the T212 [21] with 2 Kbytes of on chip memory and slower links. This was enhanced into T222 [21] and T225 [21] which have 4 Kbytes of on chip memory.

The third transputer production started with the IMS T800 [21,13]. It is an enhanced T414 version. It includes a 4 Kbytes of on chip memory, Floating point coprocessor, and four serial links. The T800 has been improved giving the T805 [21] with extra instructions and hardware pins for extra functions. This latter was enhanced into T801 [21] which includes a fast memory interface.

A summary of first generation transputer is given in the table given below.

	T212	M212	T222	T225	T414	T400	T425	T426	T800	T801	T805
On-chip memory (bytes)	2K	2K	4K	4K	2K	2K	4K	4K	4K	4K	4K
Floating point hardware	No	No	No	No	No	No	No	No	No	No	No
Word length (bits)	16	16	16	16	32	32	32	32	32	32	32
Number of links	4	2	4	4	4	2	4	4	4	4	4
Programmable DRAM controller	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Overlapped acknowledge	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Disk interface	No	Yes	No	No	No	No	No	No	No	No	No

The second transputer generation is known as the IMS T9000 [25,42]. From the user's point of view, this generation is similar to the previous one except that it is more powerful. The general architecture is the same, with a processor, a floating point unit, on chip memory, a programmable memory interface, and four communication links.

The T9000 represents a significant advance on Txxx transputers, the processor is designed to run ten times the speed of the T805. Its frequency reaches 50 MHz. The T9000 has an additional 81 instructions compared to the T805. A new feature of the T9000 is that communication between processes may take place along virtual channels that can be mapped on a single physical link. The T9000 can map up to 64 thousand virtual channels on a single link. The mapping is handled by a virtual channel processor (VCP) integrated with the processor.

CHAPTER 5

THE MULTIPLE SCALE SIGNAL MATCHING ALGORITHM

5-1 Introduction :

The aim of the Multiple Scale Signal Matching (MSSM) algorithm [26,38] is to establish the discrepancies measurement between two images, one of them being the reference.

The MSSM algorithm is based on the elastic matching theory [46]. Under this latter, one of the objects (images) is assumed to behave as an elastic material, the other serving as a reference. The matching process is accomplished after both the reference and the object (images) have been globally aligned by involving geometrical transformation such as rotation, translation, and scaling. After a global alignment, the object (image) is deformed like a piece of rubber, without tearing or folding. Deformation proceeds step-by-step so that the elastic object matches the reference.

This approach finds applications in medicine, geography, and in recognition of 3D shapes in general. This chapter is devoted to the description of the Multiple Scale Signal Matching algorithm.

5-2 General description :

The aim of the Multiple Scale Signal Matching algorithm is the measurement of the discrepancies between two images, one of them being the reference. Figure 5.1 provides the software architecture for the MSSM algorithm. Horizontally, the algorithm can be seen as consisting of a sequence of octave-separated frequency channels from the lowest frequency channel of σ to the highest one $\sigma/2^N$.

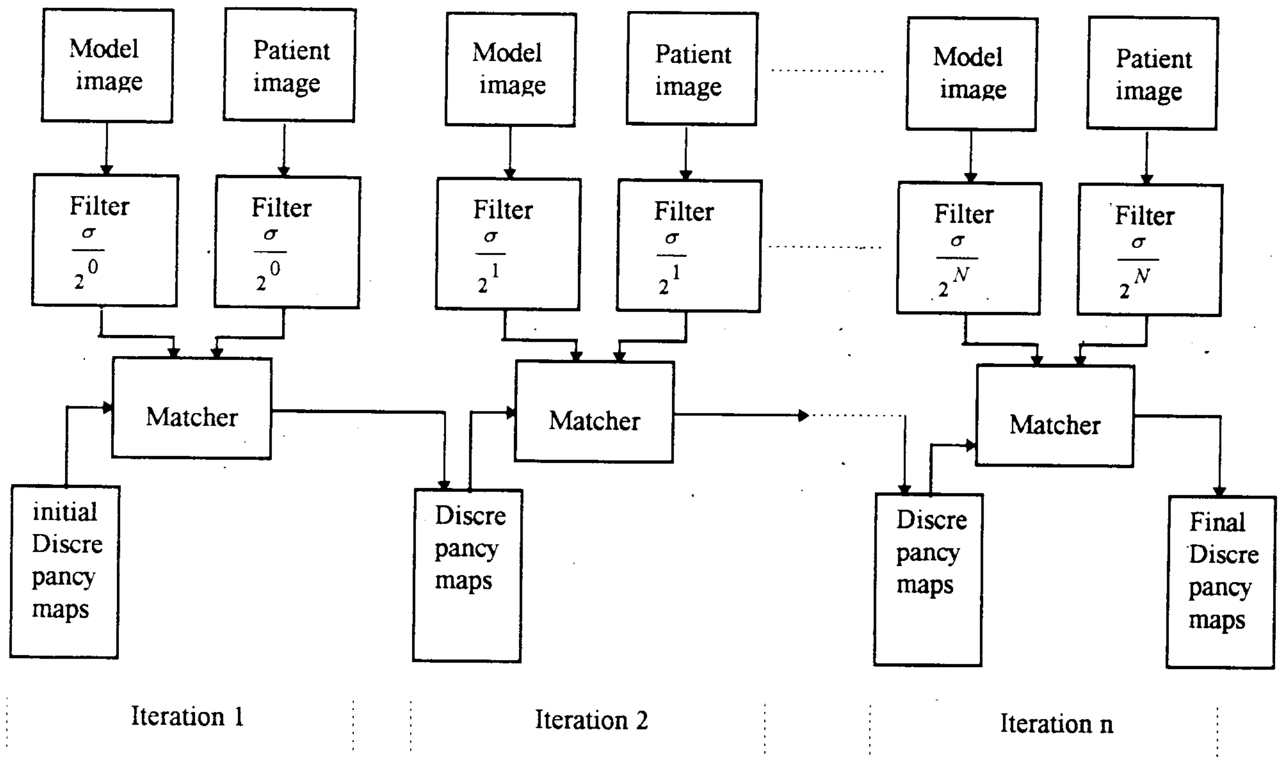

Model image
Patient image
Filter $\frac{\sigma}{2^0}$
Filter $\frac{\sigma}{2^0}$
Matcher
initial Discre pancy maps
Iteration 1
Model image
Patient image
Filter $\frac{\sigma}{2^1}$
Filter $\frac{\sigma}{2^1}$
Matcher
Discre pancy maps
Iteration 2
Model image
Patient image
Filter $\frac{\sigma}{2^N}$
Filter $\frac{\sigma}{2^N}$
Matcher
Discre pancy maps
Final Discre pancy maps
Iteration n

Figure 5.1 : Software architecture of the MSSM

At the input of the algorithm are two globally matched images and the output is a pixel-by-pixel continuous measure of horizontal and vertical discrepancies.

Basically, each of the MSSM channels consists of two main stages: a filtering stage and a matching stage. The filtering stage involves the filtering of the two images with a Difference of Gauss (DOG) filter [34]. The result is fed to the matching stage. This latter starts with an initial discrepancy estimate (i.e. (0,0)) and outputs a horizontal and a vertical discrepancy maps.

5-3 The filtering stage:

Edges, boundaries, or contours occur in physical aspects of the image and are represented by intensity changes. The aim of the filtering stage is to detect these intensity changes.

Several methods have been developed to that extent. One of these methods is known as the Gradient based method. The intensity changes can be detected by computing the first derivative of the image function. Consider a function f(x) which represents 1D intensity changes as shown in the Figure 5.2.(a). If f(x) is changing, a change in intensity is indicated by a peak in its first derivative as it is illustrated in Figure 5.2.(b)

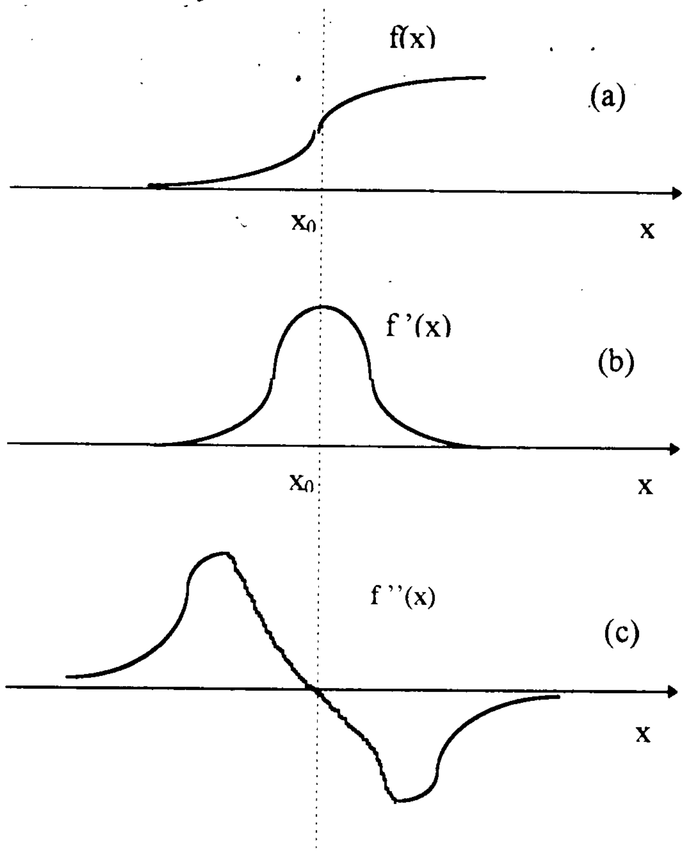


Figure 5.2: Zero-crossing.

The generalization of f '(x) to a 2D function that represent the intensity changes of an image is the gradient of f(x,y) given by:

$$\nabla f(x,y)=\frac{\partial f(x,y)}{\partial x}i_x+\frac{\partial f(x,y)}{\partial y}i_y \qquad \text{Eq 5.1}$$

where i_x is the unit vector in the x-direction and i_y is the unit vector in the y-direction.

Another method takes the second derivative of the intensity function and detects the intensity changes at the zero-crossing, or at the points where the second derivative f "(x) changes its sign as it is illustrated in Figure 5.2. (c).

The extension to 2D gives the laplacian equation as:

$$\nabla^2 f(x,y)=\frac{\partial^2 f(x,y)}{\partial x^2}+\frac{\partial^2 f(x,y)}{\partial y^2} \qquad \text{Eq 5.2}$$

Due to the fact that some edges are small and local in nature and others are large and coarse, Marr [34] stated that the intensity changes in an arbitrary image may take place at a wide range of frequencies. Thus, the direct application of the first or second derivative to the function image would not be the optimal method by which all the intensity changes can be detected. He developed a method to detect all the intensity changes arguing that two basic ideas underlay the intensity changes detection : (1) the intensity changes occur at different scales in an image and so their optimal detection requires the use of an operator that responds to several different scales; (2) a sudden intensity change gives rise to a peak or trough in the first derivative or, equivalently, to zero-crossing in the second derivative as illustrated in the Figure 5.2. The most satisfactory operator fulfilling the two above conditions is a gaussian shaped filter which has an impulse response given by:

$$h(x,y)=e^{-\frac{(x^2+y^2)}{2\pi\sigma^2}} \qquad \text{Eq 5.3}$$

where σ represents the standard deviation or the blurring parameter.

The choice of the gaussian filter is motivated by the fact that it can be tuned to a scale or frequency band at which the intensity changes are to be detected and because it is smooth and localized: a smoother h(x,y) is less likely to introduce any changes that are not present in the original shape, a more localized h(x,y) is less likely to shift the location of intensity changes.

The intensity changes can be detected from the smoothed images by using one of the methods discussed in the previous sections. In the Multiple Scale Signal Matching technique, the second partial derivative is applied in order to detect the intensity changes, by looking for the zero-crossing points using the difference of gauss filter. The mathematical representation of the filtering process is :

$$\nabla^2[f(x,y)*h(x,y)] \qquad \text{Eq 5.4}$$

It can noticed that the Difference of Gauss (DOG) filter consists of two parts, a gaussian part and a derivative part. The gaussian part is used to blur the images to the scale or band of frequency at which the intensity changes are to be detected, while its derivative part is used to detect the zero-crossing.

Thus, the filtering of the two pair of images can be achieved, first by convolving the two images with gaussian distribution function and second by taking the second derivative of the convolved functions. In the first step, the images will be blurred to a degree determined by the standard deviation sigma (σ).

The mathematical representation for the patient image (P) and the model image (M) is :

$$f_S(x,y), x=0,1,\ldots\ldots\ldots\ldots X, y=0,1,\ldots\ldots\ldots\ldots Y$$
$$S = M, or P \qquad \text{Eq 5.5}$$

The filtered image, denoted by $F_s(x,y)$ is:

$$F_S(x,y) = \nabla^2 G(x,y) * f_S(x,y) \qquad \text{Eq 5.6}$$

Where * stand for convolution.

$G(x,y)$, the gaussian distribution function, is defined as:

$$G(x,y) = \left(\frac{1}{2\pi\sigma^2}\right)e^{-\frac{(x^2+y^2)}{2\pi\sigma^2}} \qquad \text{Eq 5.7}$$

$F_S(x,y)$ can be rewritten as:

$$F_S(x,y) = \nabla^2[G(x,y) * f_S(x,y)] \qquad \text{Eq 5.8}$$

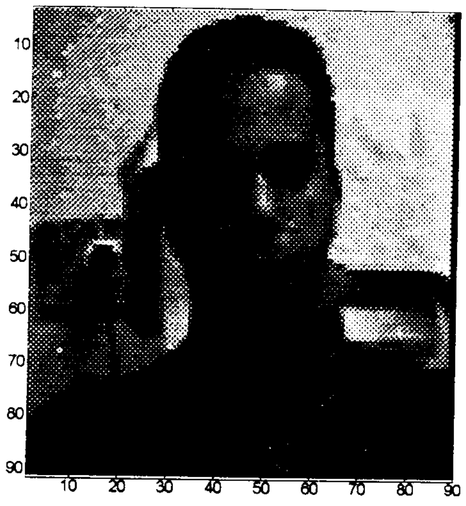
The left part between bracket of Equation 5.8 ($[G(x,y) * f_S(x,y)]$) represents the product of the distribution function and the image function. At this level the two images are blurred to a certain degree determined by the sigma (σ) value of the gaussian distribution function. This results in a blurred image $B_S(x,y)$.

In the second step the blurred images are differentiated twice. The filtered images become:

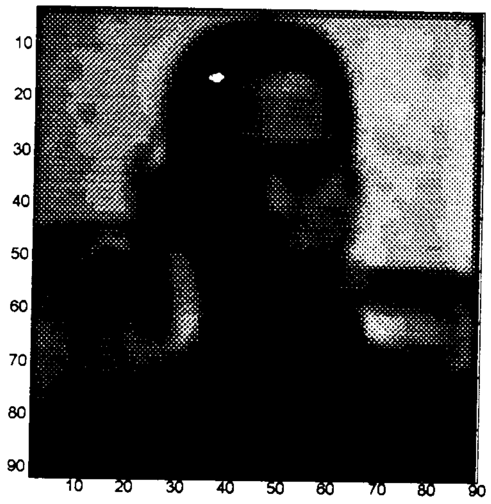
$$F_S(x,y) = \nabla^2 B_S(x,y) \qquad \text{Eq 5.10}$$

These will serve as input to the matching stage.

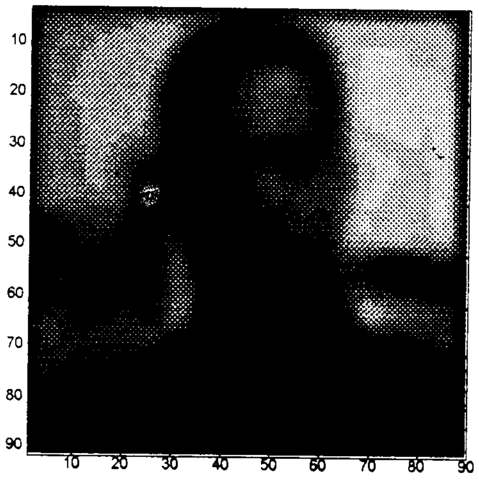
The images below show the results of smoothing (blurring) an image with gaussian distribution function with different sigma (σ=1,2,3,4,5) values.



Original image

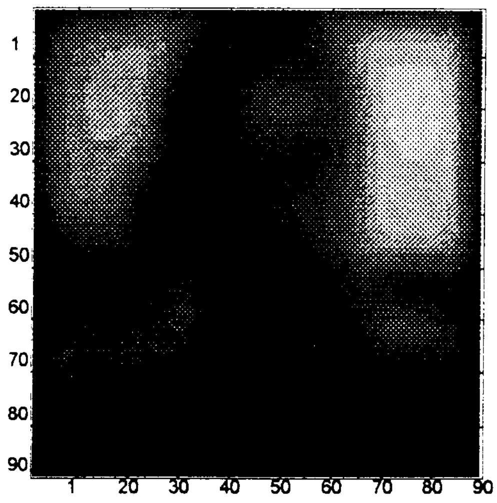


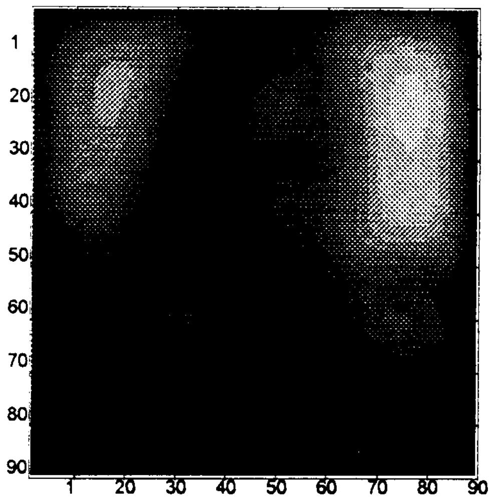
Blurred image with sigma =1



Blurred image with sigma =2

Blurred image with sigma =3

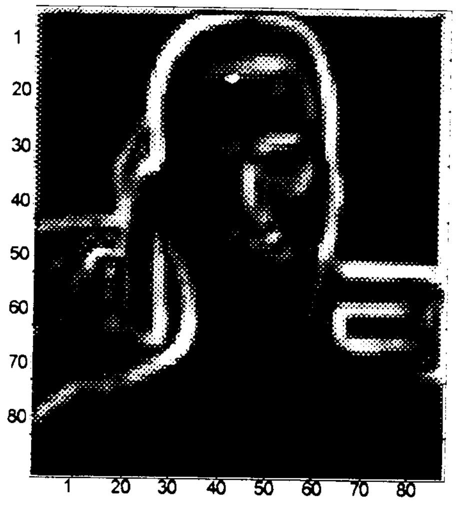

Blurred image with sigma =4


Blurred image with sigma =5

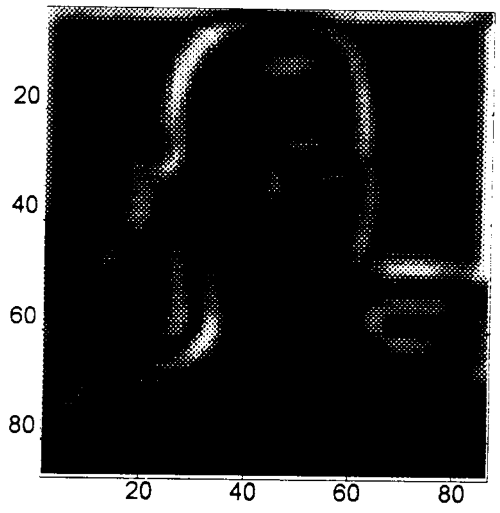
It is clear that the level of details in each of the output images varies with the sigma value. Thus, in order to take into account all intensity changes occurring at different spatial frequencies, the image must be convolved with different Difference of Gauss (DOG) filters with different sigma (σ) values.

The intensity changes in the image $f(x,y)$ will manifest themselves at the output as zero-crossing [34] in the second derivative $D^2(f*G)$. In other words, the edges can be detected by looking for zero-crossing of $D^2(f*G)$ or its equivalent D^2G*f.

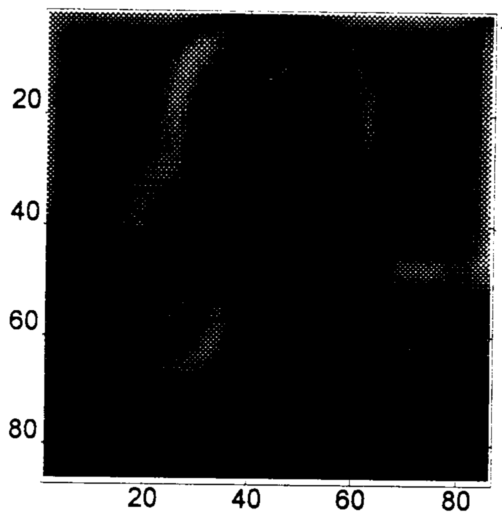
The following are examples of zero-crossing detection using DOG filters with different sigma (σ) values.



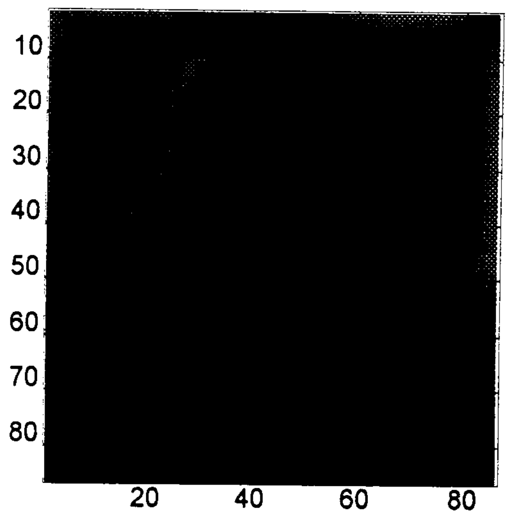
Zero-crossing detection when sigma =1



Zero-crossing detection when sigma =2



Intensity changes when sigma =3



Intensity changes when sigma =4

The number of iterations in the MSSM algorithm represents the frequency channels at which the intensity changes are to be detected. In fact this technique is based on a multichannel model based on human vision [35].

5-4 THE MATCHING STAGE :

Matching is a process in which two existing representations are put into correspondence. Having a model as reference and some input data from a sensor, the purpose of the matching process is to interpret the input data in terms of the reference.

The matching process has to be performed into two stages. First the model and the input data have to be globally aligned. This may involve translation, rotation, and scaling, so as to bring both the model and input data into an approximate correspondence. The second stage is concerned with matching each basic element of the first object with an area around the corresponding basic element on the other object. This technique is based on the elastic matching concept [46]. With the elastic matching approach, one of the two objects that are to be matched is assumed to be behave as an elastic material, the other serving as a reference. Then by deforming the elastic object like a piece of rubber without tearing or folding, its shape can be changed so that it matches the reference.

The matching stage of the MSSM algorithm is concerned only with the second stage of the matching process. This means that the input to the MSSM algorithm is two globally aligned images. The MSSM matching stage proceeds by matching one pixel image to another in the model image by looking for local matches between a pixel in the first image and a small area in the second one which is defined by a window size (w). The window (w) has its center corresponding to the matched pixel and its boundaries are placed on the neighborhood of the corresponding matched pixel (see Figure 5.3 [2]). The window size determines how much the matched pixel can be shifted from its corresponding coordinates in the model image.

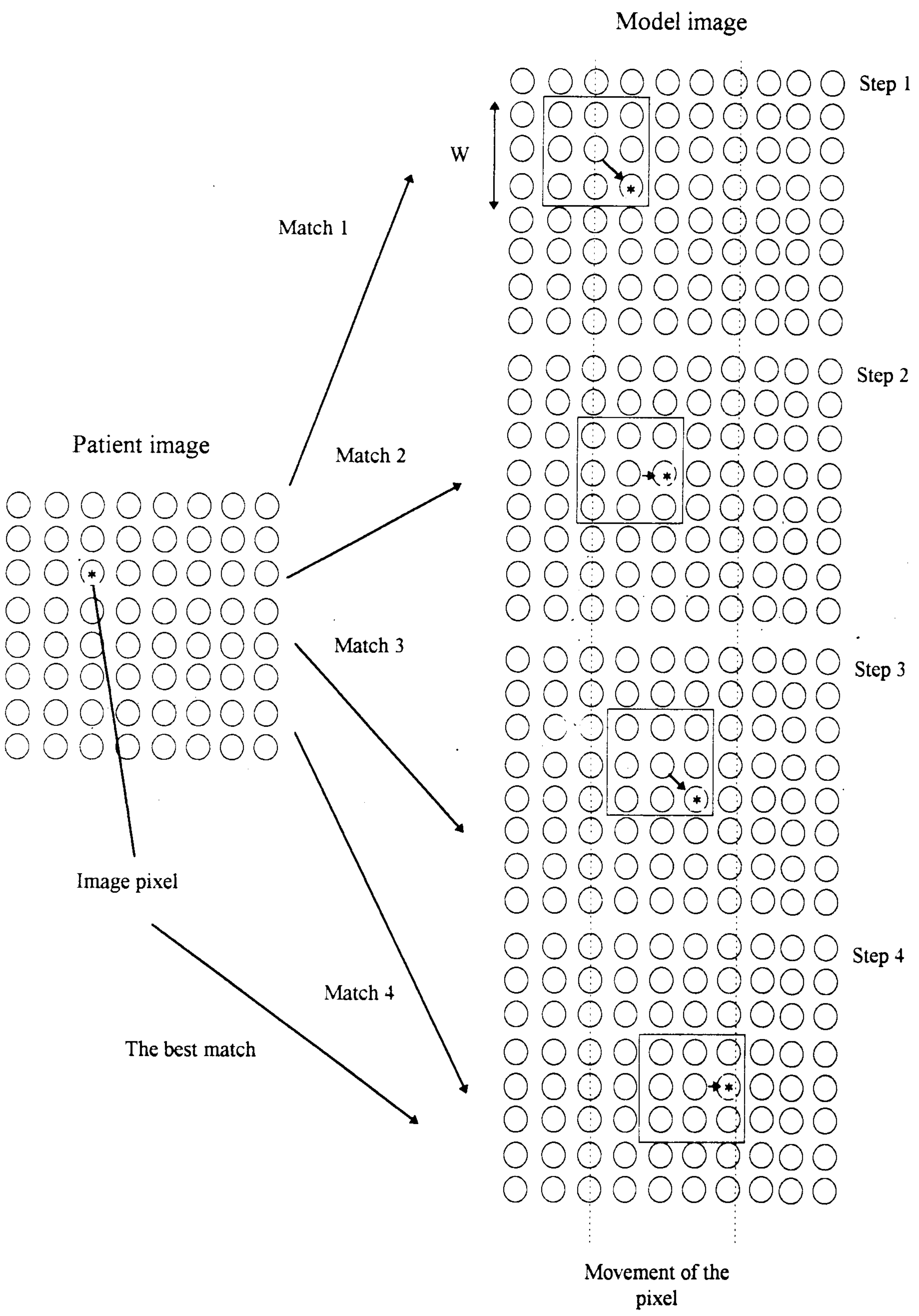


Figure 5.3 : Matching technique.

The process starts from an initial discrepancy map i.e. (0,0), and then to each iteration a discrepancy map is generated. The latter serves as initial discrepancy map for the next one. The final discrepancy map contains the corresponding coordinates of each pixel of the patient image on the model image. Once the matching process is accomplished, the correspondence between the model and the matched image is established. Therefore, any knowledge in the model can be applied directly to the matched image. For example, in the medical field as each pixel in the model image can be already known which tissue it represents, each pixel in the matched image can be classified according to the type of tissue it represents.

CHAPTER 6

SEQUENTIAL DESIGN AND IMPLEMENTATION OF THE MSSM ALGORITHM

6-1 Introduction:

The sequential design and implementation of the Multiple Scale Signal Matching (MSSM) over a single transputer system is some how necessary for the next work which deals with its parallel design and implementation over a network of transputers. The sequential implementation will serve as a basis to measure the performance of the parallel implementation.

The design of the MSSM algorithm is based on concurrent programming concepts through which the algorithm is seen as consisting of a collection of interconnected processes running concurrently in parallel on a single transputer (*T425*). With concurrent programming design methodology, each process is regarded as an independent unit. It communicates with other processes along point-to-point channels (through Memory). Using the OCCAM programming language, the communication is synchronized with message passing, avoiding the addition of any separate synchronization mechanism.

At first sight, the internal design of the process is hidden, and is completely specified by the data it sends or receives from the other processes. Each process is specified by a name, and is seen as a black box, with its communication channels represented by directed arrows that indicate the flow of data from and to the other processes.

Internally, each process can be designed as a set of communicating processes. With such approach, the algorithm design is hierarchically structured, and errors can be easily avoided. At each level the design is concerned only with a small and manageable set of processes.

6-2 Program design:

The design starts first by considering the algorithm as a system in which processes are represented by oval boxes, and communication channels by directed arrow lines. At first sight, the MSSM algorithm can be considered as a single process with two channels from which the required data transfer can be provided. These channels are connected to another process which serves as an interface between the MSSM application and its environment. For illustration, a block diagram shown in Figure 6.1 is used to give the top level representation of the program design.

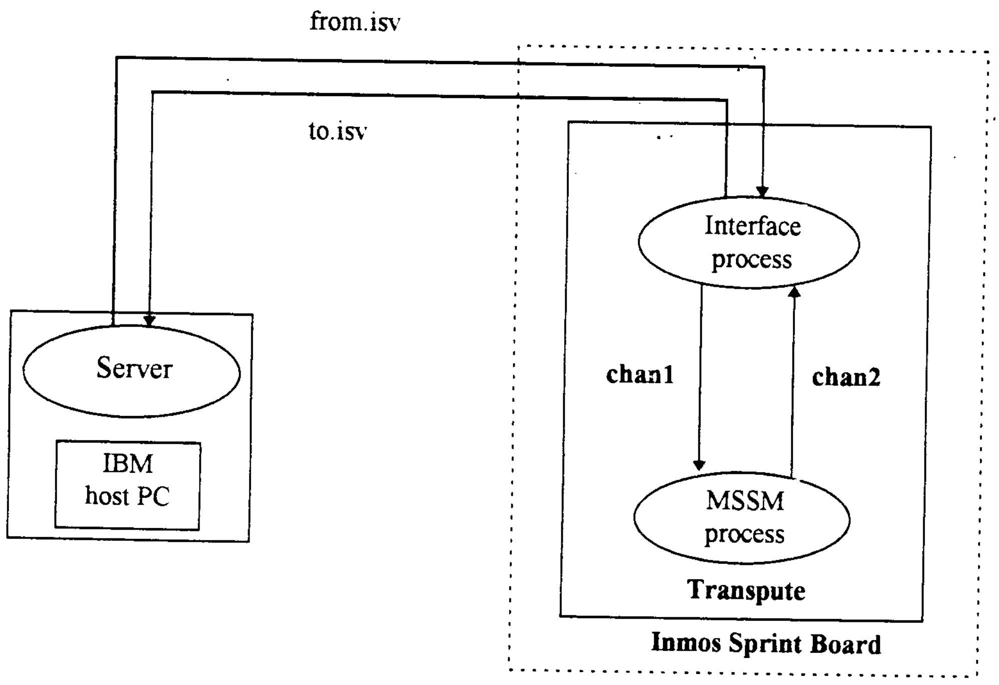


Figure 6.1 : MSSM top level design block diagram.

Each process and channel is specified by a name on the diagram. At this level a top level *OCCAM* program can be written. The two processes (INTERFACE and MSSM processes) shown in the above block diagram are designed to be executed in parallel. The only items they share are the channels between them (chan1, chan2). These latter are declared in the outer scope of the main program as shown below.

Main program

```
-- chan1 is seen as an output channel by the INTERFACE process and as an input channel by the
MSSM process.

-- chan2 is seen as an input channel by the INTERFACE process and an output channel by the
MSSM process.

-- from.isv and to.isv channels are used to connect the INTERFACE process to the server
program (SP) running on the host IBM PC

-- The KS and SS channels are used to allow the INTERFACE process to access the terminals
(keyboard, screen)

CHAN OF REAL32 chan1 :
CHAN OF INT  chan2 :
CHAN OF SP from.isv,to.isv :
CHAN OF KS keyboard :
CHAN OF SS screen :
PAR
  INTERFACE (from.isv,to.isv,keyboard,screen,chan1,chan2)
  MSSM (chan1,chan2)
:
```

In this top level design, the two processes are independent, and as long as they agree on the form of data to pass between them, the next design level can proceed by treating them independently.

6-3 The interface process:

The *INTERFACE* process is designed to handle the input and output (I/O) communication between the MSSM process and the SERVER program on the host IBM PC. *The INTERFACE* process uses two channels to communicate with the *SERVER* program in order to access the filling system on the host IBM PC. The first channel "from.isv" is used as an input from the server program and the second one (*to.isv*) is used as an output to the *SERVER* program.

Both "*from isv*" and "*to isv*" are software OCCAM channels that are mapped on the transputer's physical link0.in, link0.out respectively. Two more channels termed "*chan1*" and "*chan2*" are used by the interface process to communicate with the MSSM process. These two channels are not mapped on the transputer's hardware links, both of them are software channels. Thus, the communication between the MSSM process and the *INTERFACE* process is accomplished through the transputer on chip memory.

The *"SERVER"* program is used here as an interface between the transputer on board and the terminal, screen and the filling system. The *INTERFACE* process is the one which reads the files containing the model and patient images through the *SERVER* program using the *"from.isv"* input channel. Then it sends them to the MSSM process through the output channel *"chan1"* in which the two images will be treated. After the MSSM process completes its task, two discrepancy maps are generated, which have to be communicated to the interface process through its output channel *"chan2"* . The *INTERFACE* process, in turn has to communicate the generated discrepancy maps to the *SERVER* program through the output channel *"to.isv"* in order to save the generated discrepancy maps in the filling system.

In the *INTERFACE* process, two more channels must be defined (*"KS", "SS"*). The purpose of these two channels is to permit the user to interact with the system through the keyboard and screen.

Since the *INTERFACE* process is the one that handles the input/output communication between the *SERVER* program and the MSSM process, six procedures are to be designed. Each procedure is written in a separate fold denoted by " ... " and compiled using a separate compilation utility denoted by " SC ". The *INTERFACE* process OCCAM code can be written as follows:

INTERFACE PROCESS

```
PROC INTERFACE ( CHAN OF SP FROM.ISV,TO.ISV,CHAN OF KS KEYBOARD,CHAN
          OF SS SCREEN,CHAN OF REAL32 chan1,CHAN OF INT chan2)
... SC INPUT
... SC READ.MD
... SC READ.PT
... SC SEND
... SC RECEIVE
... SC WRITE
... LIBRARIES
... DECLARATIONS

 SEQ
    INPUT (FROM.ISV, TO.ISV,  KEYBOARD, SCREEN,NUM_FILTERS, SIZEX, SIZEY, SIGMA)
    READ.MD (FROM.ISV, TO.ISV, KEYBOARD, SCREEN, Md, SIZEX,SIZEY)
    READ.PT (FROM.ISV, TO.ISV, KEYBOARD, SCREEN,Pt, SIZEX,SIZEY)
    SEND (chan1, Md , Pt, NUM_FILTERS,SIZEX,SIZEY,SIGMA)
    RECEIVE (chan2, XMAP, YMAP, SIZEX, SIZEY)
    WRITE  (FROM.ISV, TO.ISV, KEYBOARD, SCREEN, XMAP, YMAP, SIZEX, SIZEY)
:
```

The "**INPUT**" procedure permits the user to input the number of filters (NUM_FILTERS), the size of the images (sizex, sizey) to be processed, and the blurring parameter (SIGMA). The OCCAM code of the "**INPUT**" procedure is given below:

INPUT PROCEDURE

```
PROC INPUT (CHAN OF SP from.isv, to.isv, CHAN OF KS keyboard, CHAN OF SS
                        screen, REAL32 NUM_FILTER,SIZEX,SIZEY, SIGMA)
...Declarations
...LIBRARY
...Body
   SEQ
    ---- input the number of filters
    ks.read.echo.char(keyboard,screen,chart)
    ks.read.echo.real32 (keyboard,screen, chart, NUM_FILTER)
    ----input the number of image rows
    ks.read.echo.char(keyboard,screen,chart)
    ks.read.echo.real32 (keyboard,screen, chart,SIZEX)
    ----input the number of image columns
    ks.read.echo.char(keyboard,screen,chart)
    ks.read.echo.real32 (keyboard,screen, chart,SIZEY)
    ----input the blurring parameter.
    ks.read.echo.char(keyboard,screen,chart)
    ks.read.echo.real32(keyboard,screen, chart,SIGMA)
:
```

The "**READ.MD**", and the "**READ.PT**" are written to perform the read operation of the model and patient images from the filling system on the IBM PC. The input parameters to these procedures are the two channels *"from.isv"* and *"to.isv"* by which the access to the filling system is performed, the *"KS"* channel used by input *" READ "* procedure through which the access to a file can be simulated by the keyboard, the *"SS"* channel used by the *"WRITE"* procedure by which the access to the screen is achieved, an array (Md, or Pt) of dimension N by N where the image is to be stored, and the size of the images (sizex, sizey). The OCCAM code of the "**READ.MD**", procedure is shown below.

READ PROCEDURE

```
PROC READ.Md( CHAN OF SP from.isv,to,isv,CHAN OF KS keyboard,CHAN OF SS
                    screen , [][] REAL32 img,REAL32 sizex,sizey)
... LIBRARY
... DECLARATIONS
...Body
  SEQ
     PAR
        so.keystream.from.file( from.isv,to.isv,keyboard, "c:\model image",bres)
        ss.scrstream.sink(echo)
         WHILE (pixel<>ft.terminated)
            SEQ
               ks.read.char(keyboard,chart)
               ks.read.real32 (keyboard,chart,pixel)
               SEQ
                  Md[i][j]:=pixel          -- or Pt(patient image)
                  j:=j+1
                  IF
                   j<=(INT ROUND sizey)
                     skip
                   j> (INT ROUND sizey)
                     SEQ
                        i:=i+1
                        j:=1
ss.write.nt(echo)
ss.write.endstream(echo)
:
```

The "**Send**" procedure is used to send to the MSSM process all the required data concerning the number of filters (NUM_FILTER), the size of the images(SIZEX,SIZEY), the blurring parameter (SIGMA) and also the received images (Md, Pt) from the *SERVER* program. The OCCAM code of this procedure is shown below

SEND PROCEDURE

```
PROC SEND (CHAN OF REAL32 chan1,REAL32 NUM_FILTER,sizex,sizey,SIGMA,
                [][] REAL32 Md,Pt)
... Declarations
   INT i,j:
... Body
   SEQ
      SEQ
         chan1 ! sizex
         chan1 ! sizey
         chan1 ! NUM_FILTER
         chan1 ! SIGMA

      SEQ
         SEQ i= FOR (INT ROUND sizex)
            SEQ j=1 FOR (INT ROUND sizey)
               chan.1! Md[i][j]
      SEQ
         SEQ i= FOR (INT ROUND sizex)
            SEQ j=1 FOR (INT ROUND sizey)
               chan1! Pt[i][j]
:
```

The "**RECEIVE**" procedure is needed for the reception of the generated maps from the MSSM process. Its OCCAM code is shown below.

RECEIVE PROCEDURE

```
PROC Receive (CHAN OF INT chan2,REAL32 sizex,sizey,[][] INT XMAP,YMAP)
... Declarations
   INT i,j:
... Body
   SEQ
      SEQ
         SEQ i= 1 FOR  ((INT ROUND sizex)-4)
            SEQ j=1 FOR ((INT ROUND sizey)-4)
               chan2?XMAP[i][j]
      SEQ
         SEQ i= FOR  ((INT ROUND sizex)-4)
            SEQ j=1 FOR  ((INT ROUND sizey)-4)
               chan2? YMAP[i][j]
:
```

Finally, the "WRITE" procedure performs the write (save) operation of the generated discrepancy maps into the filling system on the host IBM PC. The communication between this procedure and the SERVER program is established through the two channels "from.isv" and "to.isv". The OCCAM code of this procedure is given below.

WRITE PROCEDURE

```
PROC WRITE(CHAN OF SP FROM.ISV,TO.ISV,CHAN OF KS KEYBOARD,CHAN OF SS
SCREEN,[][] INT XMAP,YMAP,REAL32 SIZEX,SIZEY)

... LIBRARY
... Declarations
... Body
  SEQ
    PAR
      SEQ
        going:=TRUE
        i,j:=1,1
        WHILE going
          SEQ
            ss.write.int(fromprog,XMAP[i][j],4).
            j:=j+1
            IF
              j<=((INT ROUND SIZEY) - 4)
                SKIP
              j> ((INT ROUND SIZEY) - 4)
                SEQ
                  i:=i+1
                  j:=1
                  IF
                    i<=((INT ROUND SIZEX) - 4)
                      SKIP
                    i> (INT ROUND SIZEX )- 4)
                      going:=FALSE

        ss.write.endstream(fromprog)
      SEQ
        scrstream.fan.out(fromprog,tofile,secreen)
        ss.write.endstream(tofile)
        so.scrstream.to.file(from.isv,to.isv,tofile,"C:\x-map",bres)  -- Or YMAP
        IF
          (INT bres=0)
            SKIP
          TRUE
            STOP
:
```

6-4 The MSSM process :

Three processes are specified within the MSSM process: *"RECEIVER"*, *"TASK"*, and *"SENDER"*. They are designed to run in a sequential manner; see figure 6.2.

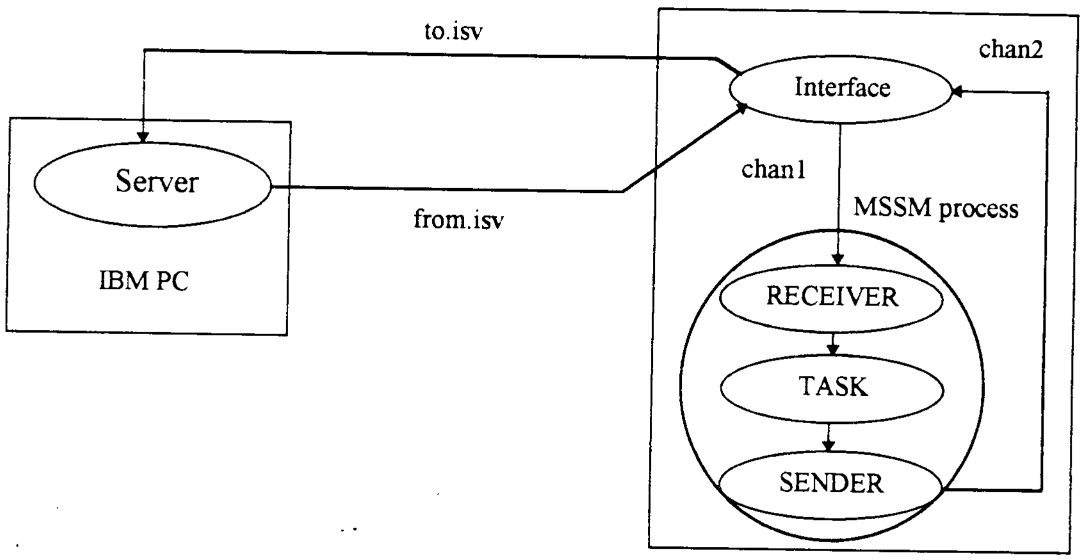


Figure 6.2 Software program configuration.

At this level each of the MSSM process is designed separately while taking into account all the items that they may share between them. The program top design level can be written with the shared parameters declared in the outer of the scope of the *MSSM* process as shown below.

<u>MSSM PROCESS</u>

```
PROC MSSM (CHAN OF REAL32 chan1, CHAN OF INT chan2 )
  REAL32 NUM_FILTER,SIZEX, SIZEY,SIGMA:
  [90 ][90 ] REAL32 Md,Pt,DOG1,DOG2, EST1,EST2:
  [90][90] INT XMAP, YMAP :
  SEQ
    RECEIVER (chan1, Md,Pt, NUM_FILTER,SIZEX,SIZEY,SIGMA,)
    TASK (Md,Pt,XMAP,YMAP, NUM_FILTER, SIGMA,SIZEX,SIZEY,)
    SENDER (chan2, XMAP,YMAP, SIZEX,SIZEY)
:
```

The three processes described above being completely independent, each process can be designed separately.

6-3.a RECEIVER process :

As shown in the block diagram of Figure 6.2, the first process *"RECEIVER"* is connected to the *INTERFACE* process via a software channel (chan1). The *"RECEIVER"* process is concerned only with the reception of data concerning the number of filters (NUM_FILTER), the size of the two images (SIZEX,SIZEY), the blurring parameter (SIGMA) and also the two images to be treated (Model, Patient). The transfer of information is established through the previously defined software channel *"chan1"*, which is connected to the *INTERFACE* process. Since the input data must be stored in shared variables in order to be used by the **Task** process, these latter are taken as input parameters to the RECEIVER procedure within the **RECEIVER** process. The *OCCAM* code of this RECEIVER procedure then can be written as shown below.

RECEIVER PROCESS

```
PROC RECEIVER (CHAN OF REAL32 chan1, REAL32 NUM_FILETER, SIZEX, SIZEY,
                        SIGMA, [][] REAL32 Md,Pt,)
  INT i,j,:
  SEQ
    SEQ
      SEQ
        chan1? SIZEX           -- RECEIVE THE IMAGE ROW NUMBER
        chan1? SIZEY           -- RECEIVE THE IMAGE COLUMN NUMBER
        chan1? NUM_FILTER      -- RECEIVE THE NUMBER OF FILTERS
        chan1? SIGMA           -- RECEIVE THE BLURRING PARAMETER
      SEQ
        SEQ i=1 FOR (INT ROUND SIZEX)    -- RECEIVE THE MODEL IMAGE
          SEQ j=1 FOR (INT ROUND SIZEY)
            chan1?Md[i][j]

        SEQ i=1 FOR (INT ROUND SIZEX)      -- RECEIVE THE PATIENT IMAGE
          SEQ j=1 FOR (INT ROUND SIZEY)
            chan1 ?Pt[i][j]
```

6-4.b The TASK process:

The second process *"TASK"* deals with the processing of the two images. The *"TASK"* process consists of four procedures: *"FILTER"* , *"BLUR"*, *"DIFGAUSS"* , and *"MATCHING"*. The four procedures are executed in a sequential manner as shown in the top design level of the *OCCAM* program shown below.

TASK PROCEDURE

```
PROC TASK (REAL32 NUM_FILETR,SIGMA,SIZEX,SIZEY[] [] REAL32 Md, Pt, XMAP,
              YMAP)
INT N:                              -- N:represents the number of iterations
[100][100] REAL32 DOG1, DOG2, ,EST1,EST2,hg:
SEQ
  N:=0
  WHILE N<= (( INT ROUND NUM_FILTERS)-1)
   SEQ
    FILTER ( SIZEX,SIZEY,SIGMA, N, hg )
    BLUR (SIZEX,SIZEY, Md, hg , EST1)
    DIFGAUSS (SIZEX,SIZEY ,EST1,DOG1)
    BLURR (SIZEX,SIZEY,Pt, hg, EST2)
    DIFGAUSS ( SIZEX,SIZEY,EST2,DOG2)
    MATCHING (SIZEX,SIZEY ,DOG1,DOG2,XMAP,YMAP)
    N:=N+1
:
```

The "*TASK*" process proceeds by first calculating the gaussian distribution function values. The "*FILTER*" procedure is designed to perform such a task. The input parameters to this procedure are: the size of the images "SIZEX, and SIZEY", the sigma value " **SIGMA** " which controls the degree level to which the two images are to be blurred, and a 2-D array "hg" of dimension N by N where the distribution function values are to be stored, and a variable "N" by which the frequency of the channel $\sigma/2^N$ can be determined.

After having calculated the gaussian distribution transfer function values, the *"BLUR"* procedure performs the convolution of the two image functions with the gaussian distribution function. The input parameters to this procedure are the gaussian distribution function values "hg(i,j)", the image to be blurred "Md(i,j)", or "Pt(i,j)", the size of the image, and an array *"EST1"*or *"EST2"* of dimension *N* by *N* where the blurred images are to be stored.

The blurred images (EST1, or EST2) will serve as input to the *"DIFGAUSS"* process which will perform the difference of gauss of the two blurred image functions. The size of the images, and the *N* by *N* array where the results are stored are taken as input parameters to the *"DIFGAUSS"* procedure.

The last procedure which performs the last stage of the Multiple Scale Signal Matching algorithm is the MATCHING procedure. The input parameters to this procedure are the difference of gauss of the model image (DOG1) and the patient image (DOG2), and two arrays of dimension *N* by *N* (XMAP, YMAP) where the X-Y discrepancy maps are to be stored.

All the procedures are written and compiled separately from the main program. Within the main program, these procedures can be called whenever one of them is needed to perform one of the task required by one of the stages of the *MSSM* algorithm.

6-4.c The SENDER process:

The *"SENDER"* process sends the two generated discrepancy maps to the *INTERFACE* process through its software output channel "chan2". Then, the interface process, in turn, will communicate them to the *SERVER* program in order to be saved in the filling system on the IBM PC. The OCCAM code of the sender process can be written as shown below:

SENDER PROCEDURE

```
PROC SENDER (CHAN OF INT chan2, XMAP, YMAP,SIZEX,SIZEY)
INT i,j:
SEQ
    SEQ
        SEQ i=1 FOR  ((INT ROUND SIZEX)-4)
            SEQ j=1 FOR ((INT ROUND SIZEY)-4)
                chan2! XMAP[i][j]

        SEQ i=1 FOR ((INT ROUND SIZEX)-4)
            SEQ j=1 FOR ((INT ROUND SIZEY)-4)
                chan2! YMAP[i][j]
:
```

At this level, as all the procedures are designed, the OCCAM code program for the whole system will have the following structure:

```
... SC Interface
... SC MSSM
... CHANNEL DECLARATIONS
PAR
   INTERF.  CE (from.isv,to.isv,keyboard,screen,chan1,chan2)
   MSSM (chan1,chan2)
:
```

6-5 Conclusion :

As it can be noticed from the above algorithmic description, the approach on which this design is based has permitted us to organize the MSSM algorithm as a set of subtasks. In this sequential design, this approach has permitted us to design a structured program through which erro could be easily avoided in the debugging process. The sequential implementation will serve as a asis for the performance measure of parallel implementation. The sequential procedures can b n as being processes or tasks that can be assigned for processors within a parallel network. The images displayed in chapter 5 have been obtained from the implementation of the algorithm in a the sequential manner described in this chapter.

At the implementation level, the experiments were carried out on images of dimension 89 by 89. The execution time of the sequential implementation with respect to the variation of the number of channels are given in table 6.1. The signification of each MSSM tasks along with their corresponding execution time obtained from the sequential implementation is summarized in table 6.2 ("sig" represents the blurring parameter).

Table 6.1

Number of channels	**Sequential execution time (min)**
1	21:22
2	29:59
3	34:18
4	36:41

Table 6.2

Task	Execution time (Min)				Associated operation
T_1	0:31				Reading the model and patient images (89 by 89).
T_2	0:04				The gaussian transfer function values calculation.
$T_{3.1}$	sig=1	0:31	sig=4	3:38	Blurring the model image.
	sig=2	1:29	sig=8	9:45	
$T_{3.2}$	sig=1	0:31	sig=4	3 :38	Blurring the patient image.
	sig=2	1.29	sig=8	9:45	
$T_{4.1}$	0:13				Applying the deference of gauss on the blurred model image.
$T_{4.2}$	0:13				Applying the difference of gauss on the blurred patient image.
T_5	0:51				Matching the two images.

CHAPTER 7

PARALLEL DESIGN AND IMPLEMENTATION OF THE MSSM ALGORITHM

7-1 Introduction:

It is well known that the Multiple Scale Signal Matching (MSSM) algorithm deals with a huge amount of data and exhibits a certain degree of computational complexity. This suggests its parallelization and its implementation on a multiprocessor system to increase its execution speed. This is highly favored by the inherent parallelism of the algorithm.

A task partition based method is used for the parallel design of the MSSM algorithm, namely the graph theoretical method. The major goal behind its parallelization would be its decomposition into subprograms or tasks that are mapped on hardware processors (transputers) in an optimal manner so that a shorter execution time will be obtained.

Experiments were carried out to measure the performance of the parallel implementation with respect to the sequential one (chapter 6). We also investigate the effect of the increase in vision channels (1 to 4) with respect to the increase in the number of processors on the computational time.

7-2 Algorithm decomposition:

Due to its software architecture (Figure 5.1), the MSSM algorithm can be decomposed into the following tasks:

- Task 1 (T_1) : Reads the model and patient images from the filling system.
- Task 2 (T_2) : Calculates the gaussian distribution function values.
- Task 3 (T_3) : Blurs the two images. For allocation clarity sake, the blurring of the model image is denoted by $T_{3.1}$ and that of the patient image is denoted by $T_{3.2}$
- Task 4 (T_4): Calculates the second derivative of the two blurred images. For the same reason as before, $T_{4.1}$ refers to the model image and $T_{4.2}$ refers to the patient image.
- Task 5 (T_5): Matches the two differentiated images from task T_4 and issues the discrepancy maps.

7-3 Parallel program design:

The MSSM algorithm has been implemented on different topologies depending on the number of transputers used (2,3, and 4). During each implementation, the number (n) of vision channels is varied from 1 to 4 to observe the effect of increasing the number of processors on the computational time. The task graph of the algorithm and the target machine are constructed. The program tasks are assigned and scheduled in such a way that a shorter execution time will be obtained. Such an objective can be achieved by assigning heavy communicative tasks on the same processor and using a task duplication technique as discussed in Chapter 3. The procedures involved in the task allocation of the MSSM program application are designed according to the following algorithmic strategy:

1- All the ready tasks (that have no predecessors) are identified and put in a ready list and ordered according to their execution time.

2- All the non ready tasks (that have predecessors) are identified and classified in a waiting list according to their execution order.

3- As long as the ready list is not empty, do the following:

3-1- Obtain as many tasks from the ready list as there are available processors.

3-2- Allocate the tasks obtained from step 3-1 on the available processors.

3-3- When all the predecessors of a task in the waiting list are executed, their successors are added to the ready list.

7-3-1 Topology 1 (two transputers) :

The target machine graph is given at the outset. It consists of two transputers T_{R1} and T_{R2} connected as shown in Figure 7.1.

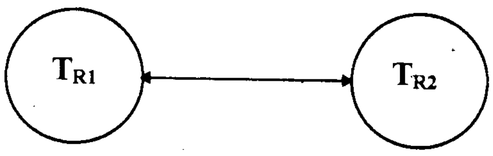


Figure 7.1: Target graph 1.

- **3.1.a Task allocation for n=1 :**(2 transputers)

The MSSM algorithm is performed in a single pass as shown in the task graph 1 of Figure 7.2. From its inspection T_1 and T_2 have no predecessors and can be put in the ready list (R.L.); see Figure 7.3.a. The remaining tasks are put in the waiting list (W.L). As there are two available processors (T_{R1}, T_{R2}), we can readily assign the two tasks T_1 and T_2 to T_{R1} and T_{R2} respectively.

Upon completion, both $T_{3.1}$ and $T_{3.2}$ are ready to be executed and are transferred from the waiting list (WL) to the ready list (RL); see Figure 7.3.b.

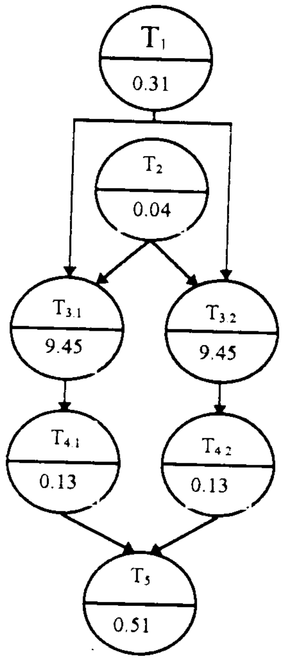


Figure 7.2 : Task graph 1

At this level, noticing the communication time delay of those tasks, we can assign them on different processors thus achieving a better starting point. When these two tasks are completed, $T_{4.1}$ and $T_{4.2}$ are added to the ready list (Figure 7.3.c) and are executed by T_{R1} and T_{R2} respectively. Finally, task 5 is ready (Figure 7.3.d) and is allocated to T_{R1}.

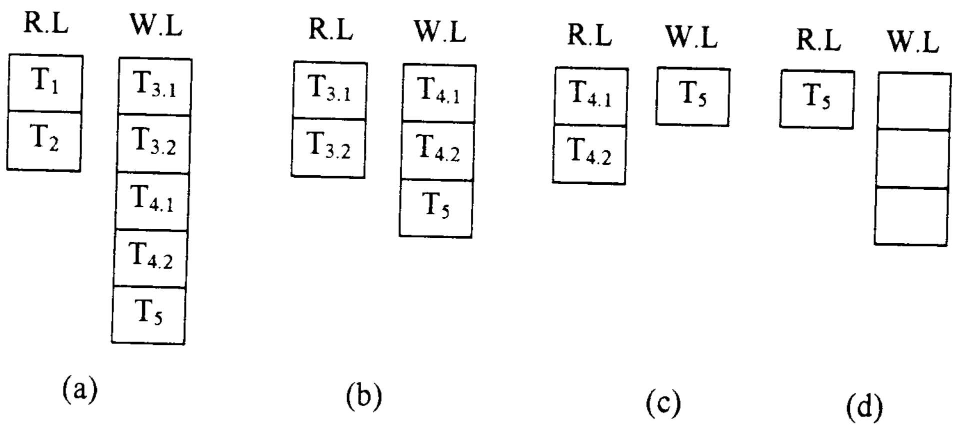


Fig 7.3 : MSSM task assignment for n=1.

Based on the previous discussion, the Gantt chart is constructed as shown in Figure 7.4.

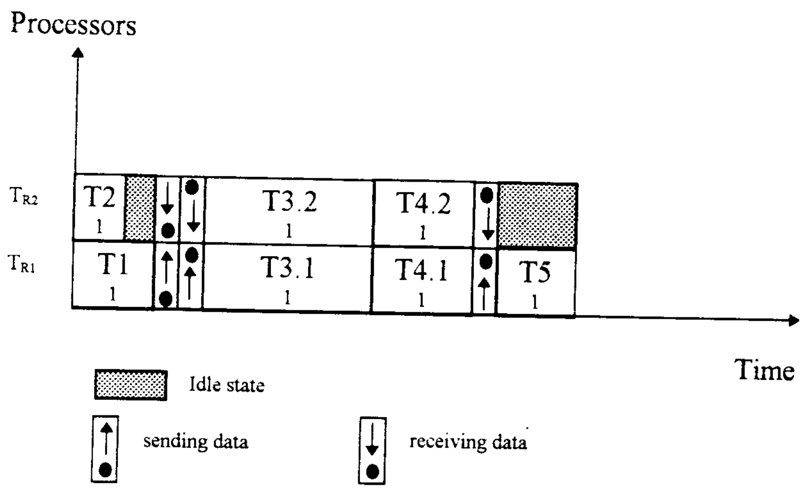


Fig 7.4 : Gantt chart 1.

- **3.1.b Task allocation for n=2 :**(Two transputers)

In this case, the MSSM is iterated twice. By inspecting the task graph shown in Figure 7.5, it can be noticed that the tasks T_1 and T_2 (both of the 1st and 2nd iterations) have no predecessors. Thus they can be considered as ready for execution. Since only two transputers are available, the tasks T_1 and T_2 of the 1st iteration are assigned to T_{R1} and T_{R2} respectively. T_2 having a much smaller execution time than T_1, we can launch the execution of task T_2 of the 2nd iteration at the end of execution of T_2 of the 1st iteration. At this level, a problem appears due to the nature of task T_2 of the MSSM algorithm. Tasks T_2 of the 1st iteration and 2nd iteration act on the same data variables, so if T_2 of the 2nd iteration is executed just after that of 1st iteration, the data integrity of this latter will be lost. This can be avoided by generating an array for each iteration that calls for a task and an additional amount of memory space (not negligible in our case).

Consequently, we can observe that the task graph in Figure 7.5 does not give all the information about tasks interdependency when the algorithm is iterated. To remedy to such a conflicting

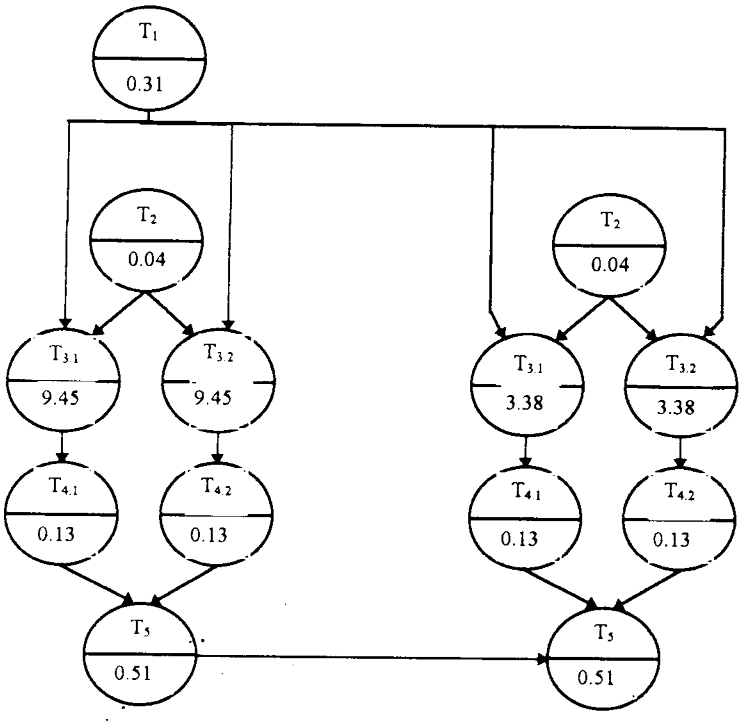


Figure 7.5 : Task graph 2.

situation, all the tasks that share the same data variables must be identified. The task precedence relationship that results must be localized and represented by dashed arrows as shown in Figure 7.6.

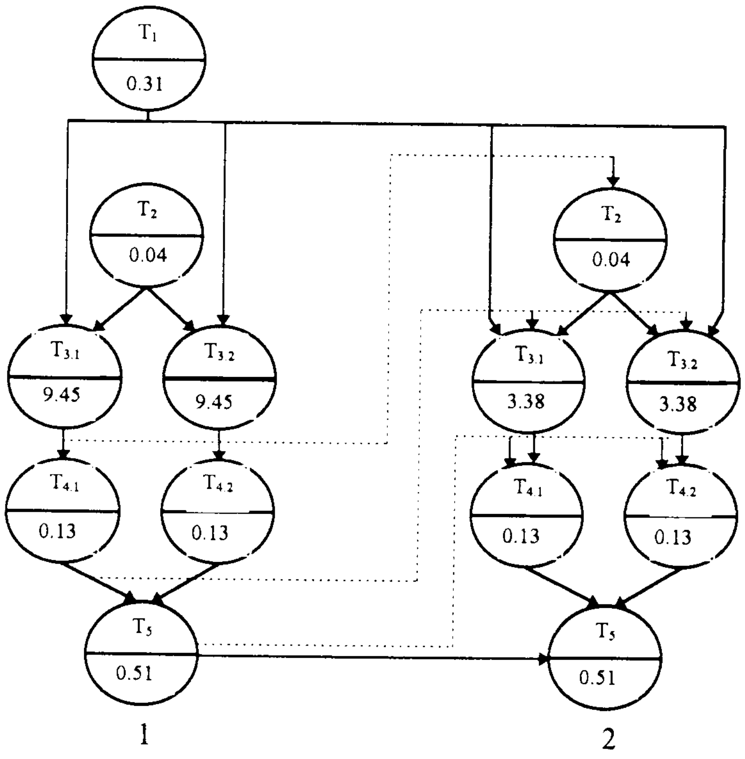


Figure 7.6 : Task graph 3.

Now, the MSSM task assignment can proceed as follows: The 1st iteration can be processed as in the first case for n=1 and postponing the ignition of the task T_2 of the 2nd iteration that is no longer ready till the completion of tasks $T_{3.1}$ and $T_{3.2}$. However, following the task execution order of the 1st iteration, $T_{4.1}$ and $T_{4.2}$ are executed just after $T_{3.1}$ and $T_{3.2}$. After this operation, T_{R1} is busy with the processing of T_5 and T_{R2} is free and consequently can receive task T_2 of the 2nd iteration. Since T_2 takes less time than T_5, tasks $T_{3.1}$ and $T_{3.2}$ of 2nd iteration are ready to be executed when T_2 reaches its finishing point. At this point only one processor is available, namely T_{R2}. Task $T_{3.2}$ is assigned to the free processor.

In the meantime, T_5 is completed and T_{R1} gets free, but task $T_{3.1}$ can not start till it gets data from T_2 (see task graph 2 in Figure 7.6). Therefore T_{R1} remains idle till the termination of $T_{3.2}$ to get data from T_{R2}; see Gantt Chart 2 in Figure 7.7. Then, T_{R2} resumes processing of task $T_{4.2}$ and is idle till $T_{3.1}$ and $T_{4.1}$ are executed on T_{R1} to pass data to task T_5 on T_{R1}.

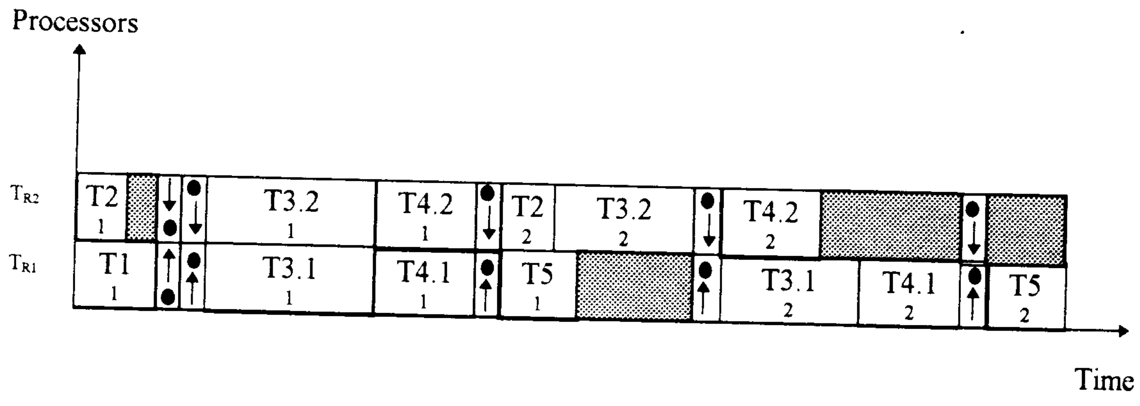


Figure 7.7 : Gantt chart 2 for n= 2

As it can be noticed, T_{R1} remains idle after T_5 of the 1st iteration waiting for $T_{3.2}$ to complete so that data transfer takes place. A solution to avoid this problem is to use the task duplication technique (discussed in chapter 3) to shift up in time task $T_{3.1}$ as shown in the Gantt Chart of Figure 7.8.

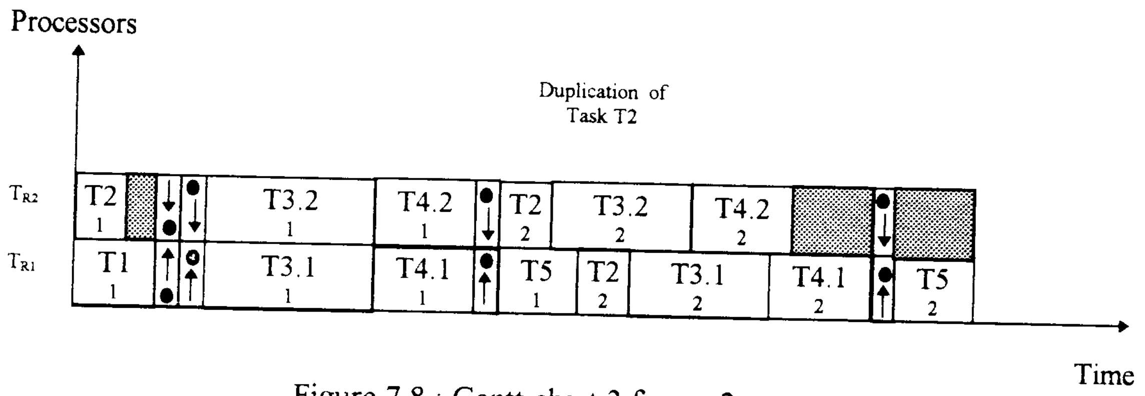


Figure 7.8 : Gantt chart 3 for n= 2

Since the execution time of task T_2 is much greater than the communication delay time, another solution based on the rendez-vous principle can be used to achieve a later starting time for task T_3 of the 2 nd iteration. After the completion of task T_2 of the 2 nd iteration, T_{R2} is kept idle till T_5 of the 2 nd iteration is terminated (Gantt Chart 4). At this point interprocessor communication (fixed in the program) is established and $T_{3.1}$ and $T_{3.2}$ can proceed simultaneously on T_{R1} and T_{R2} respectively (see Gantt Chart 4 in Figure 7.9). Thus, a best starting time can be achieved when a best rendez-vous point for both transputers is chosen. The allocation of the remaining tasks is performed as in the 1st iteration (following always the same algorithmic strategy).

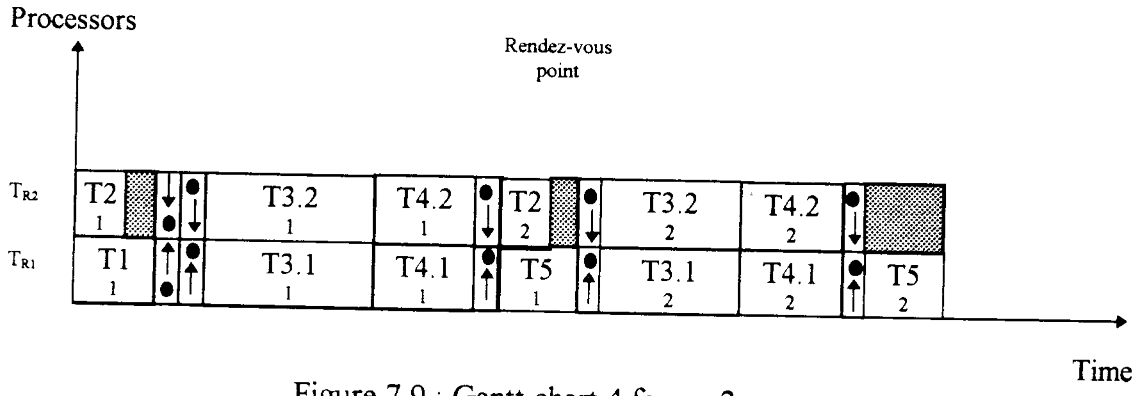


Figure 7.9 : Gantt chart 4 for n= 2

- **3.1.c Task allocation for n=3:** (2 transputers)

In a similar manner and still using the dashed arrows to represent precedence relationships between the three iterations, a task graph (Figure 7.10) is constructed and the Gantt Chart is drawn as shown in Figure 7.11.

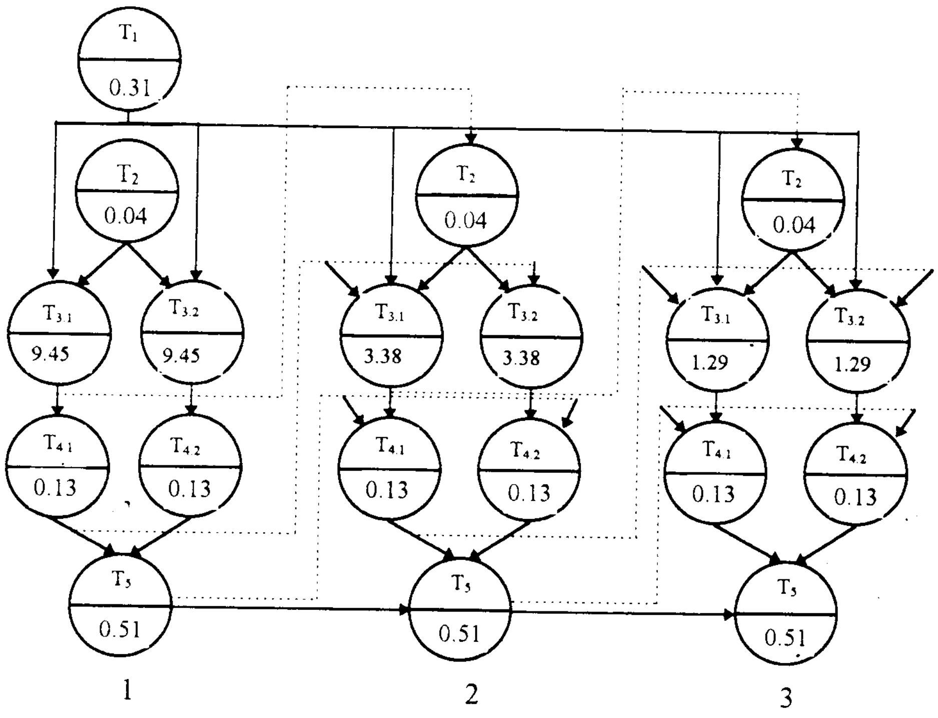


Figure 7.10 : Task graph 4.

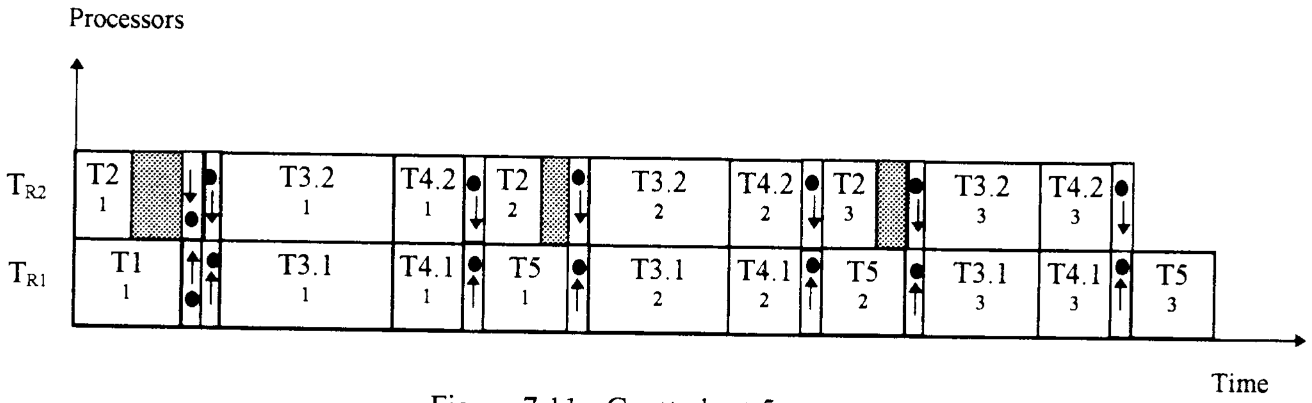


Figure 7.11 : Gantt chart 5.

- **3.1.d Task allocation for n=4:** (2 transputers)

Using the same method when the number of channels is equal to four, we obtain the task graph and the Gantt Chart shown in Figure 7.12 and Figure 7.13 respectively.

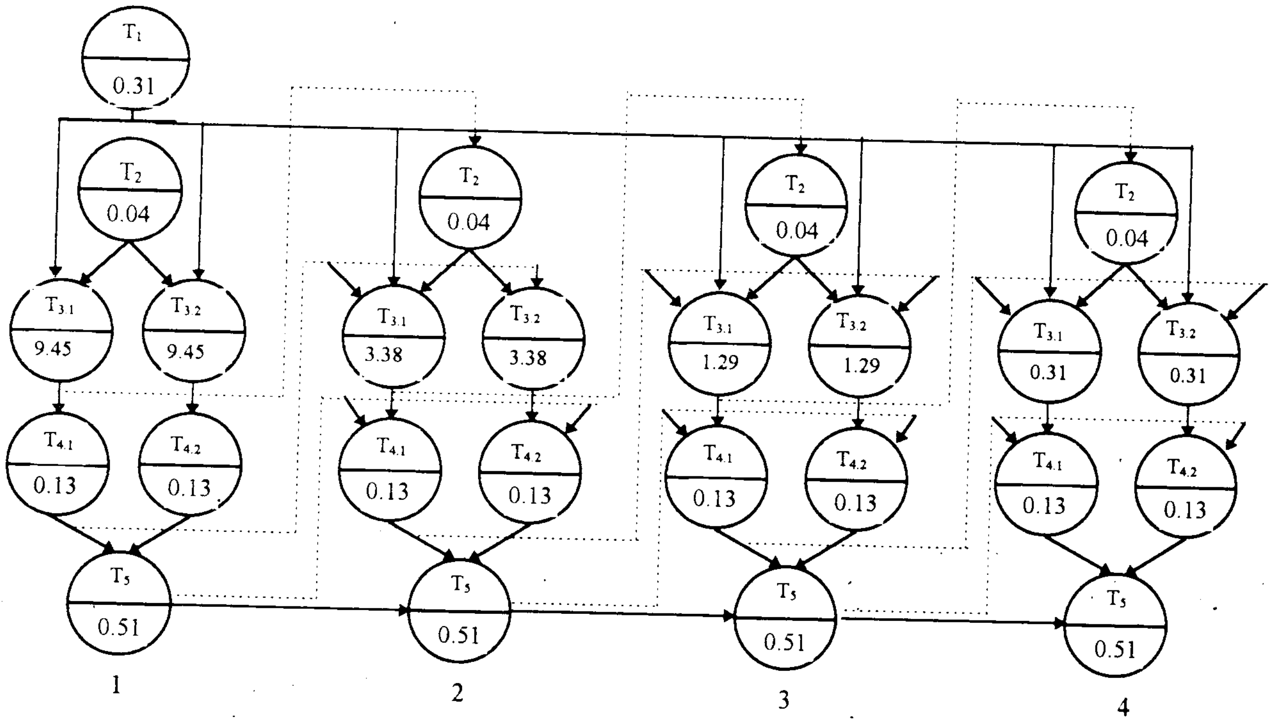


Figure 7.12 : Task graph 6.

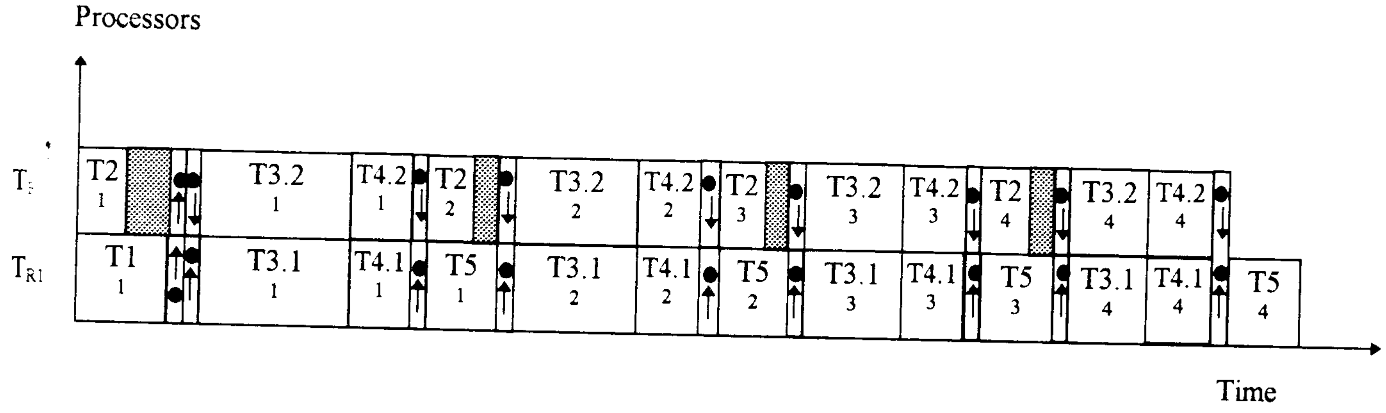


Figure 7.13: Gantt chart 6.

7-3-2 Topology 2 : (Three transputers)

The target machine graph for this topology is shown below.

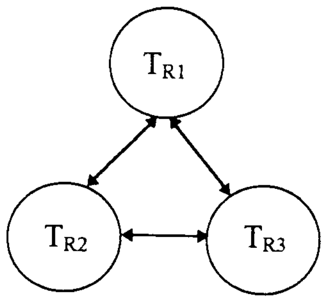


Figure 7.14 : target machine 2

- **3.2.a Task allocation for n=1:**(3 transputers)

Using the task graph shown in Figure 7.2, and the target machine graph shown above (Figure 7.14), the task allocation is the same as in the first topology and the third processor remains idle all the time. This is due to the fact that there are always no more than two tasks in the ready list as shown in Figure 7.3. The Gantt chart is shown in Figure 7.15.

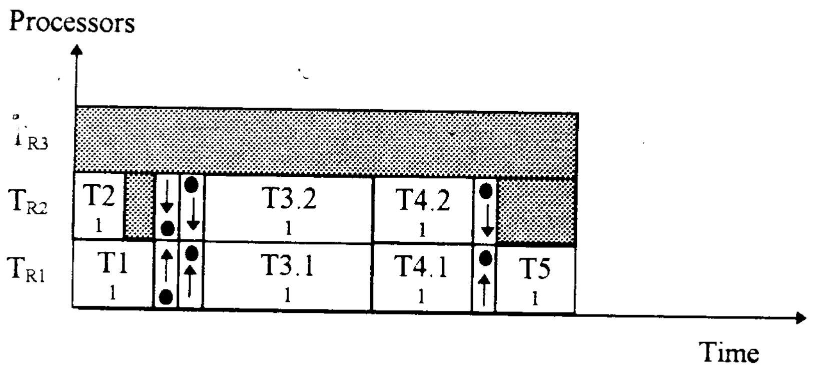


Fig 7.15 : Gantt chart 7.

- **3.2.b Task allocation for n=2 :** (3 Transputers)

Basing our allocation on the task graph of Figure 7.6 (task graph 3), we find that only tasks T_1 and T_2 of the 1st iteration are ready and task T_2 of the 2nd iteration is ready only and only if it d s not affect the data integrity of task T_2 of the 1st iteration. This can be achieved by assigning T_2 of the 2nd iteration to the third available processor T_{R3}.

At this point a very important up date has to be made to the scheduling policy concerning the meaning of a ready task.

A task is ready if it fulfills the following conditions:

- 1- *It has no predecessors.*
- 2 - *If it affects the data integrity of other tasks, it has to be run on a separate processor if available.*

Following this new definition of a ready task, the algorithmic strategy on which the task assignment is based can be updated as follows:

```
1- All the tasks that have no data dependency, and that do not depend on the termination of other
    tasks are put in a ready list.
2- All the remaining tasks are put in the waiting list.
3- While the ready list is not empty do the following:
  i- If the number of ready tasks is less than the number of available processors do:
       i.1- obtain as many tasks that have no data dependency and that may affect the data
            integrity from the waiting list as there are remaining available processors, and add them to the
            ready list.
       i.2- Allocate the ready tasks on the available processors.
       i.3- Go to step 5.
  ii- Else
       ii-1 Allocate as many ready tasks on as many available processors. The tasks are taken
            with respect to their execution order.
       ii-2 Go to step 5
5- Whenever a task reaches its finishing point, its successors are added to the ready list.
```

Following this new algorithmic strategy, the task assignment in the present case can be done as shown in the Gantt chart shown below.

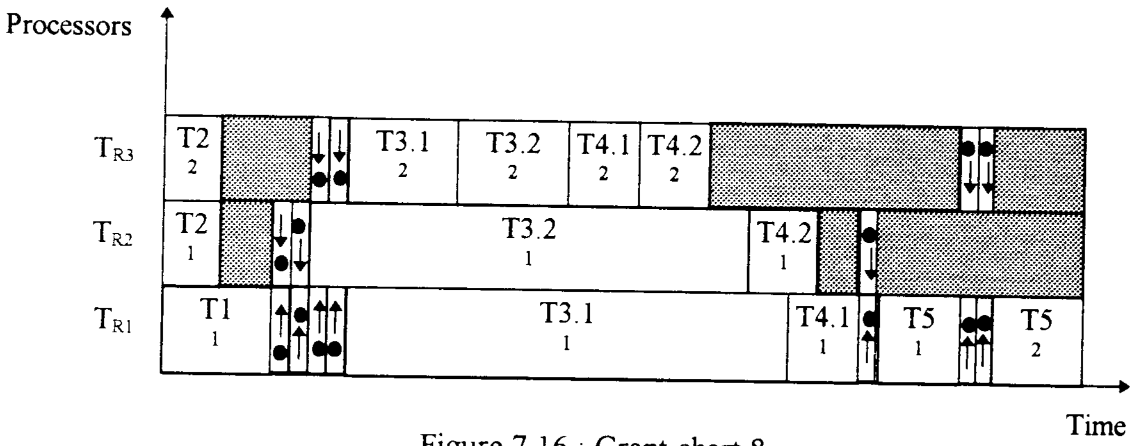


Figure 7.16 : Grant chart 8.

- **3.2.c Task allocation for n=3 :** (3 Transputers)

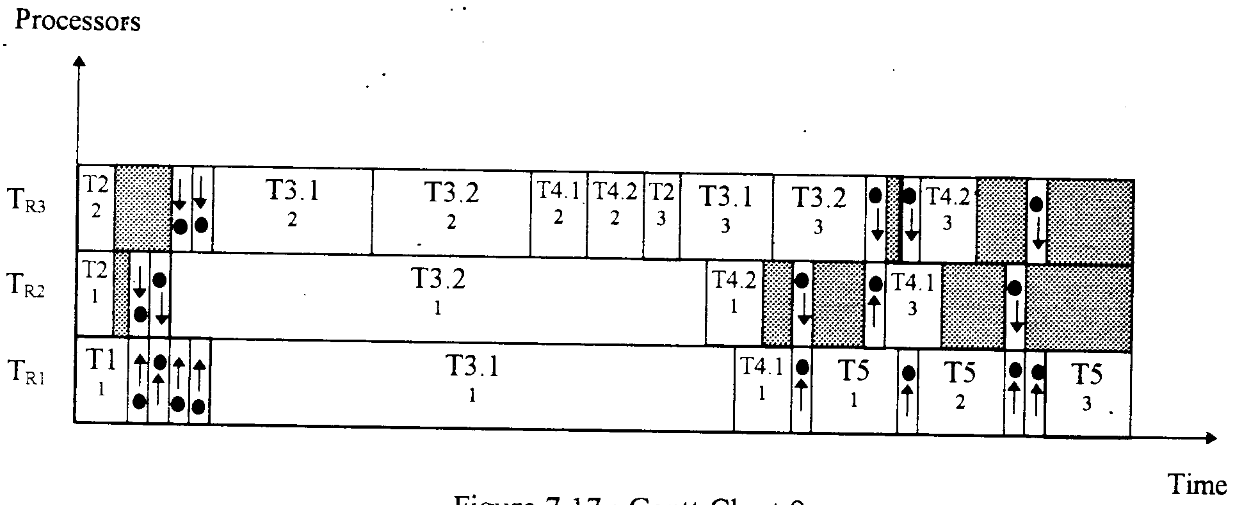


Figure 7.17 : Gantt Chart 9

- **3.2.d Task allocation for n=4 :** (3 Transputers)

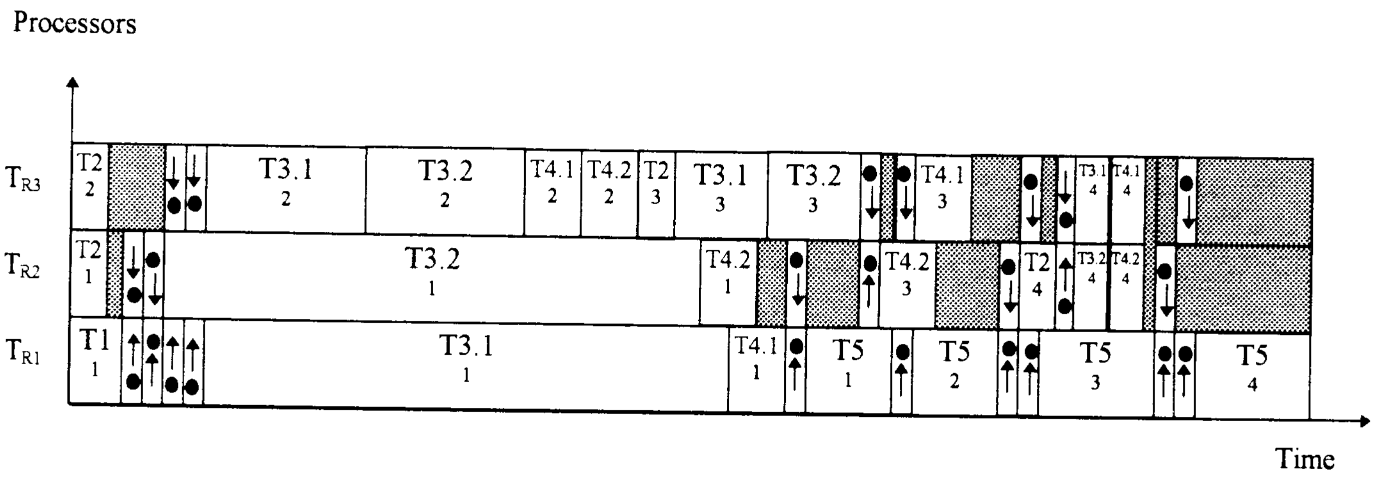


Figure 7.18 : Gantt Chart 10

7-3-3 Topology 3: (Four transputers)

The third parallel target machine consists of four transputers connected as shown in the following target machine graph.

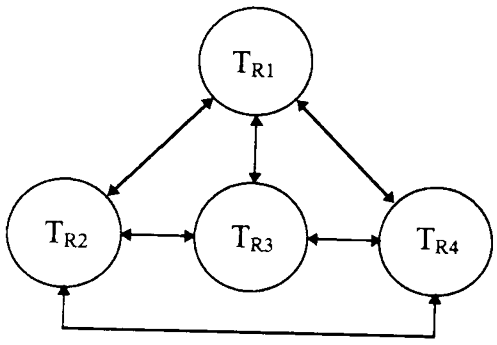


Figure 7.19 : target machine graph 3

- **.3.a Task allocation for n=1 :** (4 Transputers)

By applying the same algorithmic strategy in connection with the task graph of Figure 7.2, it can be noticed that the number of ready tasks is less than the number of available processors. Therefore, T_{R3} and T_{R4} are not utilized as shown in the Gantt chart of Figure 7.20.

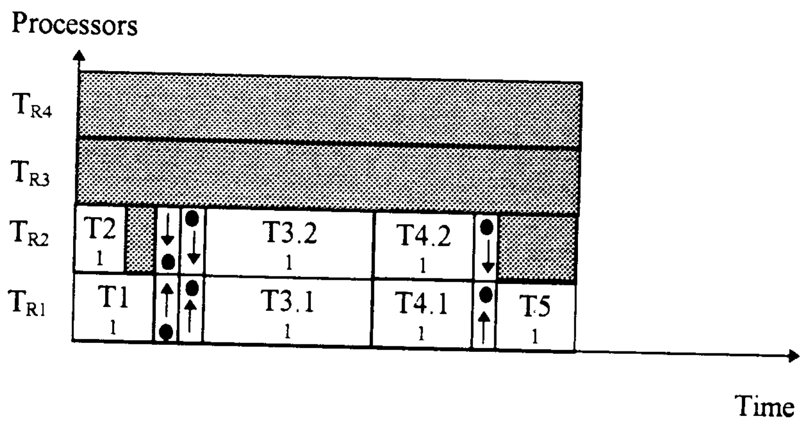


Fig 7.20 : Gantt chart 11.

- **3.3.b Task allocation for n=2:** (4 Transputers)

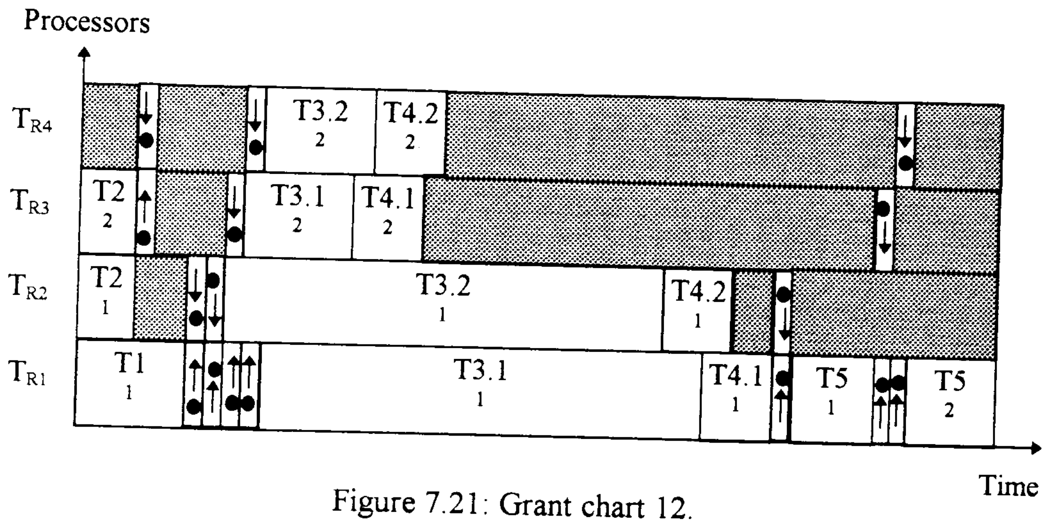


Figure 7.21: Grant chart 12.

- **3.3.c Task allocation for n=3:** (4 Transputers)

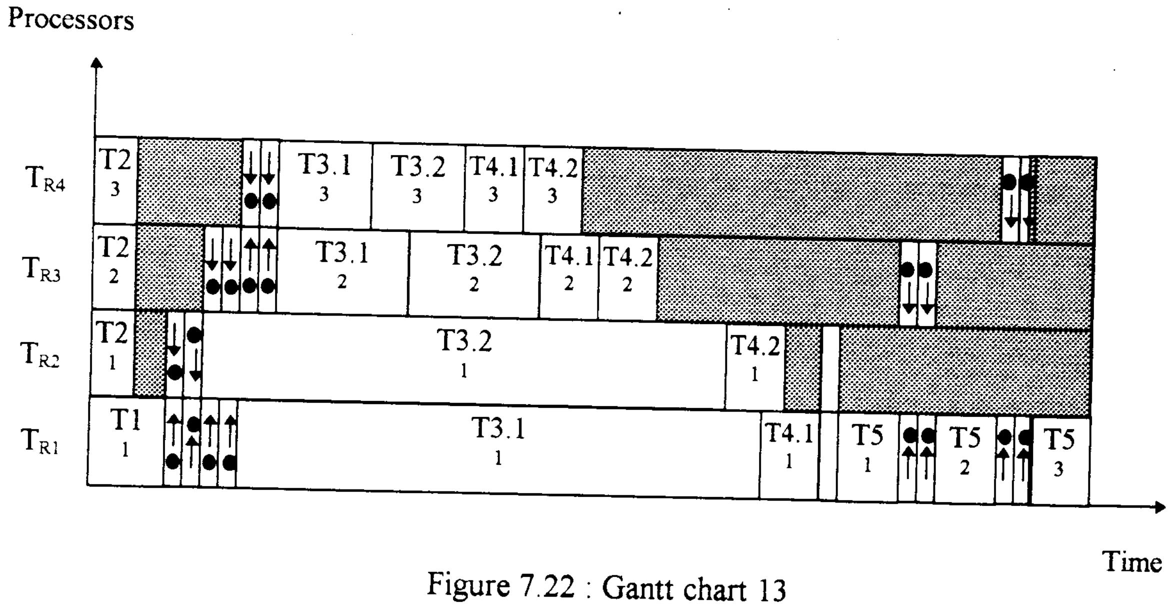


Figure 7.22 : Gantt chart 13

- **3.3.d Task allocation for n=4 :** (4 Transputers)

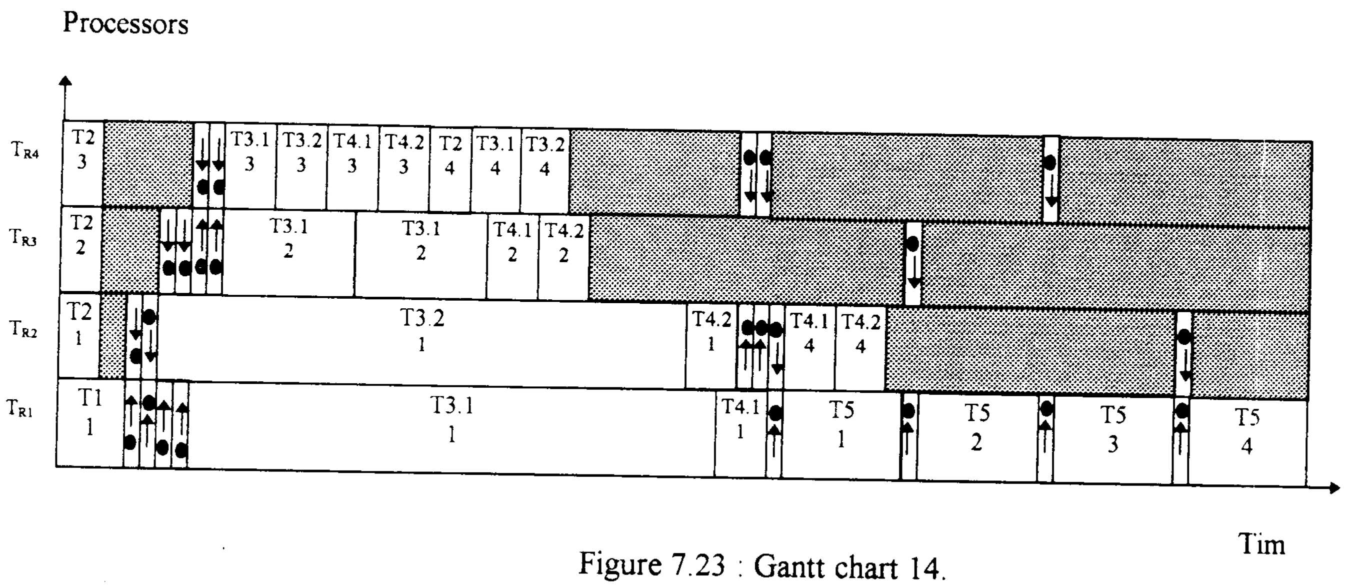


Figure 7.23 : Gantt chart 14.

7-4 Implementation:

The available hardware resources for the implementation of the MSSM algorithm consist of a number of INMOS half length PC plug in boards. These are known as SPRINT boards.

The SPRINT board comprises an INMOS T425 transputer with four serial links through which communication between a set of transputers can be established. The four transputer links are provided on the DB25 socket at the rear of the board. Each board is plugged on an IBM PC. Then through the DB25 socket a set of transputers can be connected through wires to construct a network as shown in Figure 7.24.

The required data, 89 by 89 images, are stored in the filling system of only one IBM PC (Host). The transputer (Master) which is allocated on this PC is connected to the PC bus system by link 0 through an interface circuit (C012). Thus, for this transputer, only three remaining links

(1,2, and 3) are available to be connected to the other transputers (Slaves). These latters have four links available for their connection within a network.

The programming language used for this implementation is OCCAM. The programs were developed under the transputer development system (TDS3).

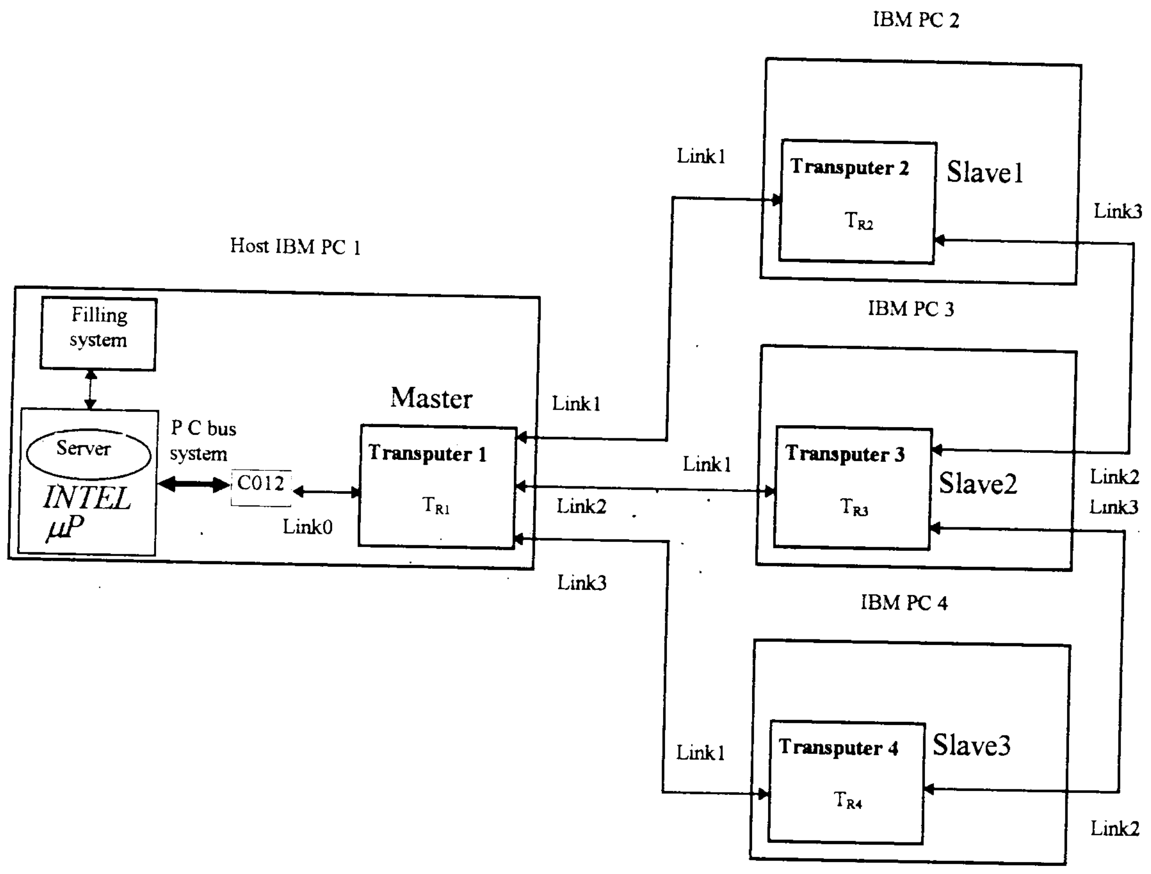


Figure 7.24 : Network configuration.

Before the tasks are allocated on the transputers, all the control signals (Reset, Analyse, and Error) of each transputer must be chained together, and connected to the Master transputer (the transputer running the TDS3 on the host IBM PC). In order for the TDS3 to boot the transputers (Slaves) and programs can be loaded on the network, it needs only to be connected to the Master transputer via its link0. The remaining transputers can be booted via any one of their links from the Master transputer or from any one that has been booted from the Master transputer.

For the parallel implementation of the MSSM algorithm, the tasks (T1 through T5) have been already designed in the sequential implementation (Reading the images (T1), The filter transfer function values calculation (T2), Blurring (T3.1, T3.2), Differentiation (T4.1, T4.2), and the Matching (T5)). The remaining work is concerned with the design of the procedures that will be used for the transfer of data between the elementary tasks of the MSSM algorithm that will be allocated on the transputers.

On each transputer, all the tasks and communication procedures are compiled using a separate compilation (SC) utility. They are put in the outer scope of the OCCAM program and are called when needed from the main program. The OCCAM program code for each implementation will have the following structure (... refers to a Fold):

1) ... SC Tasks
2) ... Libraries
3) ... Declarations
4) ... Channels placements
5) ... Main program

All the tasks and the required communication procedures allocated on a processor are compiled with a separate compilation utility and put in a fold denoted by SC. The libraries (I/O, Math, ...) used are put in the second fold. The variables and software channels are specified in the third fold. The forth fold is used to specify the mapping of the software channels on the hardware ones. Finally the fifth fold is used to receive the main program from which all the required procedures can be called.

In what follows, we will present the implementation of the first program (one channel) of the 1st topology (two Transputers). The remaining ones are similar, they differ only in the tasks allocated to the processors.

- **Topology 1** : (Two transputers)

The transputer links interconnection of this topology is shown in Figure 7.25.

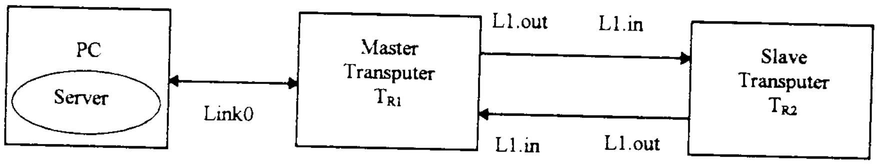


Figure 7.25: Transputer links interconnection.

- **OCCAM program code for the Master Transputer:**

From the Gantt Chart1 of Figure 7.4, it can be noticed that three communication procedures are needed for the transfer of data between the Master (TR1) and the Slave transputer (TR2). As the hardware links connections are established (as shown in Figure 7.25) and the task are allocated on the two transputers (as shown in the Gantt Chart 1 of in Figure 7.4), three communication procedures are designed to perform the transfer and the reception of data to and from the Slave transputer (TR2).The code of each one of them is given below:

SEND procedure:

This procedure is used to transmit the image to be treated by the Slave transputer (TR2). The input parameters of this procedure is a 2D array (Pt) in which the image is to be stored and a channel (Out) through which it will be sent.

```
SEND (CHAN OF REAL32 Out, [][] REAL32 Pt)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         Out ! Pt[i][j]
:
```

RECEIVE procedure (1):

This procedure is used to receive the gaussian distribution function values from the Slave transputer (TR2). The input parameters of this procedure is a 2D array (hg) in which the data values are put and a channel (In) through which these values will be received.

```
RECEIVE 1 (CHAN OF REAL32 In, [][] REAL32 hg)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         In ? hg[i][j]
:
```

RECEIVE procedure (2):

This procedure is used to receive the image that has been treated by the Slave transputer (TR2). The input parameters of this procedure is a 2D array (Dog2) in which the image will be stored and a channel (In) through which it will be received.

```
RECEIVE 2 (CHAN OF REAL32 In, [][] REAL32 Dog2)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         In ? Dog2 [i][j]
:
```

At this level, the whole OCCAM program code of the Master transputer can be implemented as shown below.

Master OCCAM code program:

```
... SC Monitor        -- this process performs task T1 (It reads the two images)
... SC Blurr          -- This process performs task T3.1.
... SC Gauss          -- This process performs task T4.1
... SC Match          -- This process performs task T5
... SC SEND
... SC REICEIVE1
... SC RECEIVE2

... Libraries

[90][90] REAL32 hg,Md,Pt, Dog1,Dog2,Xmap,Ymap:
CHAN OF REAL32 chan1,chan2 :

PLACE chan1. AT Link1.in     -- chan1 is mapped on link1.in
PLACE chan2 AT Link1.out     -- chan2 is mapped on link2.out

SEQ                               -- main program
   Monitor (from.isv, to.isv, keyboard, screen,Md,Pt)
   SEND (chan2, Pt)
   RECEIVE (chan1,hg)                 -- "hg" is a two-dimensional array
   Blur(hg,Est,Md)                    -- "Est and Md" are two-dimensional arrays
   Gauss(Est,Dog1)                    -- "Dog1" is a two-dimensional array.
   RECEIVE2(chan1,Dog2)               -- "Dog2" is a two-dimensional array.
   Match (Dog1,Dog2,Xmap,Ymap)        -- "Xmap and Ymap " are two-dimensional arrays
```

- **OCCAM program code for the Slave transputer (TR2):**

For the Slave transputer, the OCCAM code program can be written as follows: From the Gantt Chart 1 of Figure 7.4, it can be seen that three communication procedures are to be design for the accomplishment of the transfer of data with the Master transputer. These are designed as follows:

RECEIVE procedure:

This procedure is used to receive the image to be treated by the Slave transputer (TR2). The input parameters of this Procedure are a 2D array (Pt) in which the image will be stored and a channel (In) through which it will be received.

```
RECEIVER (CHAN OF REAL32 In, [][] REAL32 Pt)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         In ? Pt [i][j]
:
```

SEND1 procedure:

This procedure is used to transmit the gaussian distribution function values to the Master transputer (TR1). The input parameters to this procedure are a 2D array (hg) in which the data values are put and a channel (Out) through which the data values will be sent.

```
SEND1 (CHAN OF REAL32 Out, [][] REAL32 hg)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         Out ! hg [i][j]
:
```

SEND2 procedure:

This procedure is used to transmit the treated image to the Master transputer (TR1). The input parameters of this procedure are a 2D array (Dog 2) in which the data values are put and a channel (Out) through which the data values will be sent.

```
SEND2 (CHAN OF REAL32 Out, [][] REAL32 Dog2)
INT i,j:
SEQ
   SEQ i=1 FOR 89
      SEQ j=1 FOR 89
         Out ! Dog2 [i][j]
:
```

As this level, the whole OCCAM program code for the Slave transputer can be written as follows:

Slave OCCAM program code:

```
... SC Filter         -- this process performs task T2
... SC Blurr          -- This process performs task T2.1.
... SC Gauss          -- This process performs task T3.1
... SC SEND1
... SC SEND2
... SC RECEIVE

... Libraries

[90][90] REAL32 hg,Pt, Est,Dog2:
CHAN OF REAL32 chan1,chan2 :

PLACE chan1 AT Link1.in     -- chan1 is mapped on link1.in
PLACE chan2 AT Link1.out    -- chan2 is mapped on link2.out

SEQ                         -- main program
  Filter (hg)               -- "hg" is a two-dimensional arrays
  RECEIVE1(chan1,hg)
  SEND1 (chan2, hg)
  Blurr(hg,Est,Pt)           --  "Est,Md" are two-dimensional arrays.
  Gauss(Est,Dog2)            --  "Dog1" is a two-dimensional array.
  SEND2 (chan2,Dog2)
```

Once the MSSM tasks and the communication procedures are compiled, the following steps are followed in order to avoid the deadlock problem.

- The software channels must be mapped on the correct and corresponding physical links according to the interconnection shown in Figure 7.25.
- Both the Send and Receive procedures must agree on:

  - The type of the channels used.

- The type of data to be transferred and received.

- The size of the data to be transmitted and received.

7-5 Results :

Architecture involving 2, 3, and 4 transputers have been implemented and tested. Tables 7.1, 7.2, 7.3, and 7.4 show the variation in the number of processors for each variation in the number of channels of the MSSM algorithm. For every implementation, the time gain is calculated according to the equation $G = T_S / T_P$.

System	# of processors	Execution time (min)	Time gain
Single transputer	1	21.22	
Topology 1	2	11.22	1.89
Topology 2	3	11.22	1.89
Topology 3	4	11.22	1.89

Table 7.1: Number of vision channels =1

System	# of processors	Execution time (min)	Time gain
Single transputer	1	29.59	
Topology 1	2	16.22	1.82
Topology 2	3	12.14	2.43
Topology 3	4	12.14	2.43

Table 7.2: Number of vision channels =2

System	# of processors	Execution time (min)	Time gain
Single transputer	1	34.18	
Topology 1	2	18.38	1.85
Topology 2	3	13.06	2.61
Topology 3	4	13.06	2.61

Table 7.3: Number of vision channels = 3

System	# of processors	Execution time (min)	Time gain
Single transputer	1	36.41	
Topology 1	2	20.14	1.80
Topology 2	3	13.59	2.67
Topology 3	4	13.59	2.67

Table 7.4 : Number of vision channels = 4

From the results shown in table 7.1, it can noticed that the time gain obtained for topology 1 (with 2 transputers) is almost equal to the expected one (G=2). However, the results obtained for the remaining two topologies (with 3, and 4 transputers) reveal that no appreciable time gain is obtained although a greater number of processors is used. Obviously, this can be explained by returning to the examination of the task graph 1 of Figure 7.2 and the Gantt Chart 1, 7, and 11 from which it can be seen that due to the task precedence relation, during the task process, each time the number of ready tasks are equal or less than the number of available processors.

Tables 7.2 relative to channel 2 reveals that an appreciable time gain of 1.82, and 2.43 for topology 1, and 2 which approach the expected ones (G=2 for topology 1, and G=3 for topology 2) is obtained. However, no improvements have been obtained when moving from topology 2 to topology 3 and this is due to the sequential nature of the matching stage of the MSSM algorithm as shown on the Gantt Chart 2, 8, and 12 of Figure 7.13, 7.16, and 7.21. Furthermore, this observation is valid when the number of channels is increased to 3, and 4 as seen in tables 7.3 and 7.4.

The execution time versus the number of channels is shown in Figure 7.26, 7.27, and 7.28 for the three topologies. In Figure 7.26, we observe that the change in the execution time varies considerably when passing from 1 channel to 4 (11.22-20.14). However, we notice that for the same number of channels, this change is smaller (11.22-13.59) and identical for both remaining topologies (2,3). This time is fixed by the sequential nature of the MSSM algorithm as can be observed on the Gantt Chart 12,13, and 14.

Since the change is the same for topologies (2, and 3) including 3, 4 transputers, an optimal choice would be obviously a selection of topology 2 (With 3 transputers).

Figure 7.26

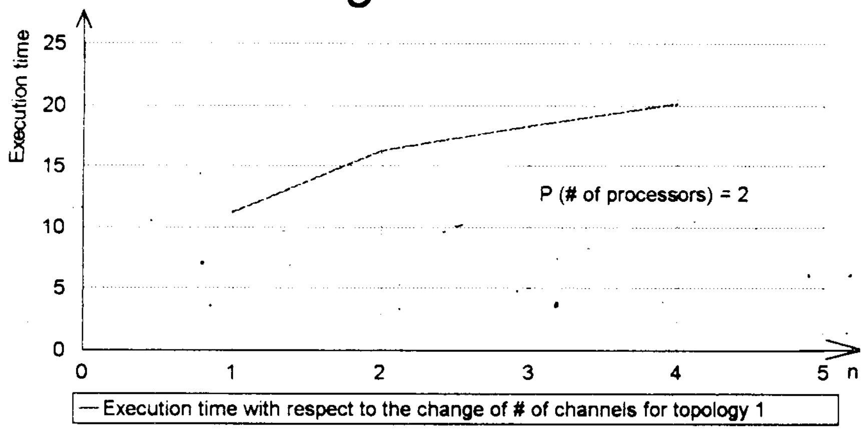


Figure 7.27

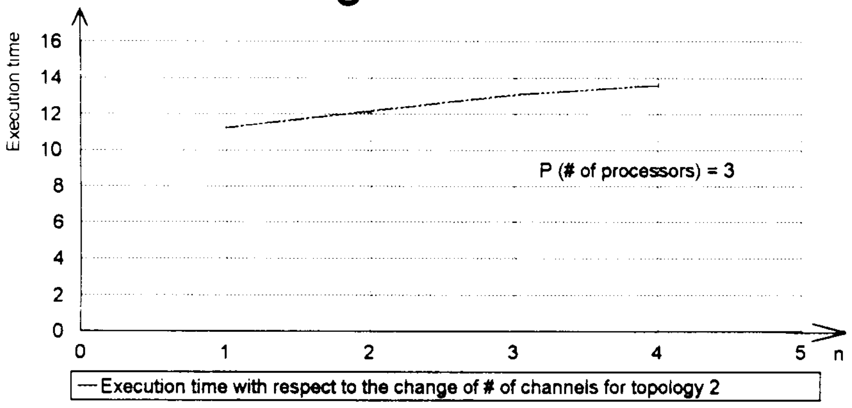
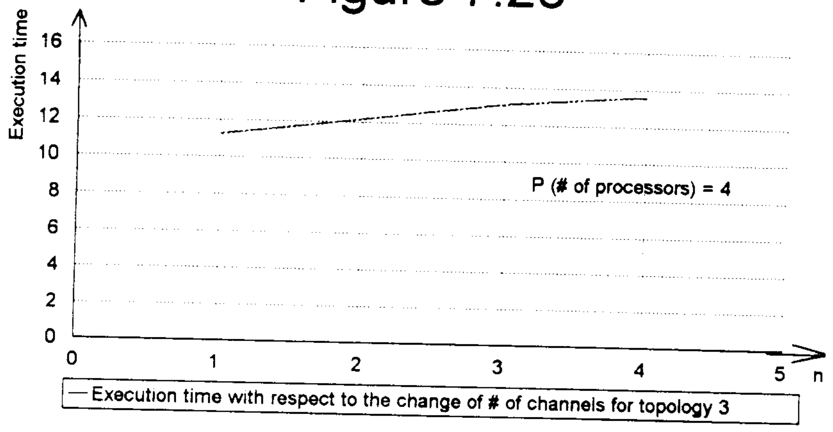


Figure 7.28



Execution time
16
14
12
10
8
6
4
2
0
0
1
2
3
4
5
n
P (# of processors) = 4
— Execution time with respect to the change of # of channels for topology 3

Conclusion :

The task-level partition approach has been used to parallelize an application algorithm, namely the Multiple Scale Signal Matching (MSSM). The result of the partition was mapped on a network of transputers. Architectures involving 2,3, and 4 transputers have been implemented and tested. The experiments were carried out to measure the performance of the parallel implementation with respect to the sequential one. The effect of the increase in the computational time when additional vision channels (1 to 4) are used with respect to the increase in the number of processors has been also investigated. The performance tests indicate a substantial improvement in speed compared --h a single processor execution. The performance analysis has permitted us to identify, among three topologies, the most suitable one for such an application, which is the one, including three tranputers. Additional processors will not increase the performance of the algorithm and this is due to the sequential nature of the matching stage of the MSSM algorithm.

At the design level, a graphical model has been used to represent the MSSM algorithm tasks as well as the data flow which represent the algorithm tasks dependency. However, due to the iterative nature of the MSSM algorithm, we perceived that it would be a mistake to always consider that the intertask dependency is due only to the data dependency. In fact a task starting ignition can be dependent on the termination of another task if both of them act on the same data. In such a case, we suggest that before the task assignment may take place, all the intertask dependencies must be present on the application task graph. Then, depending on the number of available processors, the tasks whose ignition may depend on the termination of other tasks can be considered ready to be allocated if and only if the number of processors is greater than the number of ready tasks (which have no predecessors).

Although the major goal of implementing the MSSM algorithm on a set of transputers has been achieved, there are still prospects for further improvements and extensions. This is limited only by the nature of the MSSM algorithm. To obtain other performances, two other approaches can be applied: Data-level partition and parallelization at the instruction level. The first approach splits the image in many subimages that will be treated individually by each processor of the parallel machine. The partial results obtained from each processor are then gathered to construct the whole treated image. This method requires a large number of processors and leads also to increased communication delays. It gives also birth to boundary problems at subimage edges. The second method which is based on the parallelization of operations included in a single task of an algorithm is characterized by analysis complexity and requires a larger number of processors.

REFERENCES

[1]- A.B. Fontaine, F.Barrand/A.Rawsthorne, "80286 and 80386 Microprocessors New PC Architectures", MacMillan, 1989.

[2]- A. Bouklachi, "Parallelization of the Multiple Scale Signal Matching algorithm", Report, University of leads, School of computer studies, 1991.

[3]- A.J. Von De Goor, " Computer Architecture and Design ", Addison-Wesley Publishing Company, London, 1989.

[4]- Burns, A, " Programming in OCCAM 2", Addison-Wesley, Workingham, 1988.

[5]- C.A.R Hoare, " Communicative Sequential Processes", Prentice-Hall. Inc, 1985.

[6]- Charles E. Ginarc and Veljko M. Milutinovié, " A Survey of RISC Processors and Computer of Mid-1980s", IEEE Trans, Computer, Vol 20, No 9, 1987.

[7]- Collin Whitby-Strevens, " The Transputer", The Proceeding of the 12th International Symposium on Computer Architecture, pp. 292-300, 1985.

[8]- David A. P and Carlo H.S, " A VLSI RISC", IEEE, Computer, Vol 15, No 9, pp. 8-18, 1982.

[9]- David B. Skillicorn," A Taxonomy For Computer Architecture", IEEE Trans, Computer, Vol 21, No 11, pp. 46-57, 1988.

[10]- Ernest Hirch, " Les Transputers: Application a la programmation concurrente", EYROLLES,1990.

[11]- George R. Desrochers. " Principle of Parallel and Multiprocessing", McGraw-Hill Book Company, 1987.

[12]- G.Lindhorst, A. Andrson, and D. Dahms, " Programming the 68040", BYTE, pp. 121-128, August 1991.

[13]- H.D. Mark, D. shepherd, and R. shepherd, " The IMS T800 Transputer", IEEE Trans, Micro, pp. 10-26, 1987.

[14]- H.Lu, M.J Carey. " Load-Balanced Task allocation in Locally Distributed Computer Systems.". IEEE Trans, Computer, Vol 22, No2, pp. 1037-1039, 1986.

[15]- H.S.Stone, " Multiprocessor Scheduling with the Aid of Network Flow", IEEE Trans, Software Eng, Vol.SE- 3, pp. 85-93, 1977.

[16]- H.T. King, " Network- Based Multiprocessors: Redefining High Performance Computing in the 1990's ", Decernial Caltech Conference on VLSI Pasadena, California, 20-22, pp. 1-18, March 1989

[17]- Ian Watson and Jonh Gurd," A Practical Data Flow Computer", IEEE Trans, Computer, Vol 15, No 2, pp. 51-57, 1982.

[18]- INMOS Limited, " OCCAM 2 Reference Manual", Prentice Hall: Hemel hempstead, 1988.

[19]- INMOS Limited, " The Transputer Development System", Prentice Hall, 1990.

[20]- INMOS Limited, " IMS B008", User guide and reference manual, 1990.

[21]- INMOS limited, " The Transputer Data Book", 1st edn. Nov 1988. INMOS ltd, Bristol.

[22]- J.B.Dennis, "The Varieties of Data Flow Computers", The proceeding of the first International conference on distributed Computing Systems, pp. 430-439, 1979.

[23]- J.c.Heudin and C. Panetto, " RISC Architecture", CHANPMAN and Hall, 1992.

[24]- J.D. Nicoud, A.M.Tyrrell, " The Transputer T414 Instruction Set", IEEE Trans, Micro, 1989.

[25]- Jeremy Hilton, Alan Pinder, " Transputer Hardware and System Design", Prentice Hall, 1993.

[26]- Jin Z.P, Mowforth, P, " A Discrete Approach to Signal Matching", Technical Report TIRM -89-036, The Turning Institute, Glasgow, Scotland, October 1988.

[27]- John Uffenbeck, "The 8086/8088 Family: Design, Programming and Interfacing", Prentice Hall Inc, 1986.

[28]- Jones, G, Goldsmith M, " Programming in OCCAM 2", Prentice Hall, Hempstead, 1988.

[29]- K.E. Batcher, " Design of a Massively Parallel Processor", IEEE Trans on Computers, Vol.C-29, No 9, pp. 386-840, Sept 1980.

[30]- Kemel Elfe, " Heuristic Model of Task Assignment Scheduling in Distributed Systems", IEEE Trans, Computer, Vol 15, No 6, pp. 50-56, 1982.

[31]- Li, k and Hudah, P, " Memory Coherence in Shared Virtual Memory Systems", ACM Trans, Computer systems. Vol 7, No 4, pp. 321-359, 1989

[32]- M.Akil, E.Dujardin," Parallelisation des transformations torphologiques des images sur Reseau de Transputer", La letter de transputer et des calculateurs distribués. pp. 7-19, 1992.

[33]- Mark Ainsworth, " SMT101 ' SPRINT' Board", User manual. Sundance Multiprocessor Technology Ltd, 1991.

[34]- Marr, D. " Vision: A Computational Investigation into The Human Representation and Processing of Visual Information", San Francisco, CA, W.H. Freeman, 1982.

[35]- Martin D. Levine, " Vision in Man and Machine", McGraw-Hill Book Company, 1985.

[36]- Martti.J.Forsell, " Are Multiport Memory Physically Feasible ?", Computer Architecture News. Vol 22, No 5, pp. 3-10, 1994.

[37]- Per Stenstrom, " Reducing Contention in Shared Memory Multiprocessors", IEEE Trans, Computer, Vol 21, No 11, pp. 26-37, 1988.

[38]- Peter H. Mowforth, Jin Zhengping, " Model Based Tissue Differentiation in MR Brain Images", Proceedings of the 5th Alvey Vision Conference, Vol 25-28, pp. 67-71, Sept 1989.

[39]- Philip H.Enslow Jr, " Multiprocessors and Parallel Processing", A Wiley-Interscience Publication, 1974.

[40]- Pierre Vincent, " Nouvelles architectures d'ordinateurs processus et systèmes d'exploitation", Éditests, 1989.

[41]- Pountain, D and May, D, " A Tutorial Introduction to OCCAM Programming", BSP professional books, London, 1987.

[42]- Pountain. D, " The Transputer Strikes Back", Byte, pp. 256-271, August 1991.

[43]- P.R.Ma, E.Y.S.Lee and M.Tsuchiya, " A Task Allocation Model for Distributed Computing Systems", IEEE Trans, Computer, Vol C-31, No 1, pp. 41-47, 1982.

[44]- Rafael C. Gonzalez, " Digital Image Processing", Addison-Wesley Publishing Company,Inc, 1987.

[45]- R.M. Russell, " The CRAY-1 Computer System", Communications of the ACM, Vol 21, No 1, pp. 63-72, January 1978.

[46]- Ruzena Bajcsy and Stane Kovacic, " Multiresolution Elastic Matching", Academic Press, Octobre 1988.

[47]- Ted G. Lewis and Hesham El-Rewini, " Introduction to Parallel Computing", Prentice Hall, 1992.

A N N E X E

Liste et composition du jury en vue de la soutenance de la thèse de magister en Ingénieur des Systèmes Elèctronique.

Par Mr CHEREF Mohamed.

PRESIDENT /: Dr B. TEDJINI BAILICHE Hacene
Docteur d'Etat (Maître de Conférnce à l'USTHB.

REPPORTEUR /: Dr HARICHE Kamel.
Chargé de Recherche à l'I.N.E.L.E.C.

EXAMINATEUR /: Dr FARAF Ahcene.
Docteur d'Etat (Maître de Conférence à l'E.N.P.

: Dr BENMOHAMED Kouider
PHD (Chargé de Cours à l'I.N.E.L.E.C).

/ Mr BOUKLACHI Abbes.
Master (Maître Assistant à l'I.N.E.L.E.C).