Ministry of Higher Education and Scientific Research
University M'hamed BOUGARA Boumerdes
Faculty of Engineering
Department of Electrical Engineering and Electronics

Report

Presented in Partial Fulfilment of the requirements of the

MAGISTER DEGREE

In Electronics Systems Engineering

Title

APPLICATION OF PARALLEL PROCESSING TO MEDICAL IMAGING

Soutenu le: 27/09/2005

By

Abbès Bouklachi

In front of the Examiner board

Pr. M. MEZGHICHE	Prof. at Univ. of Boumerdes	President
Pr. K. BADARI	Prof. at Univ. of Boumerdes	Supervisor
Dr. A. HERZELLAH	Maitre de conference at the Univ. of Boumerdes	Examiner
Dr. M. AHMED NACER	Maitre de conference at the Univ. of USTHB	Examiner

2007/2008

To my family

wife Khedidja

and children Mohamed Fouad, Mounir, Madjda, Nassim Eddine, and Abderaouf.

Acknowledgements

The author wishes to thank Professor Kamel Baddari for his supervision at the University of Boumerdes, Faculty of Science, Department of Computer Science.

Thanks to Professor Mohamed Zitouni for his encouragement and support.

Thanks to Professor K. Harriche for his supervision at the electrical engineering department.

Thanks to Professor Peter M. Dew for his supervision during the three years I have been at Leeds.

The financial support of the British Council is gratefully acknowledged.

Thanks are due to those in Parallel programming Group in Computer Studies, University of Leeds, UK for providing thought provoking environment over the few years I have spent with them.

Particular thanks are due to Doctor Terrance Fernando for his technical support and discussions.

Thanks to Professor David Hogg and to his Artificial Group which have provided the medical application and NMR data from the Leeds General Infirmary scanner.

Thanks to my students who have carried in their thesis and projects the development of many tools and software applications necessary to realise the theoretical aspects of this research work, and it would have been impossible without their hard work and efforts. Among them are: Fadli,Khodja, Laroui, mougari, Boukhobza, and Derraa.

الملخص

يعنى هذا البحث بتطبيقات البرمجة المتوازية في التصوير الطبي من خلال تطوير حلول ذات فعالية عالية لحساب بعض المعالجات التي تمر بها الصورة في أجهزة التصوير الطبي من حيث أداء التنفيذ و قابلية التسلق و الشمولية.

لقد انتقينا معالجات شديدة التعقيد و الحمولة، ثم طورنا لها حلولا ذات فعالية عالية. لقد اتبعنا في تطوير هذه الحلول على مبادئ البرمجة المتوازية التي تستغل التزامن المتضمن في الحساب. هذا من جهة و من جهة أخرى خزنا صورة عضو الإنسان في شفرات ترتب معطيات العضو المحصل عليها من أجهزة الفحص بحيث تمكننا من ربح الوقت في معالجة المناطق المتجانسة. كما اعتمدنا في بحثنا هذا على استعمال مختلف أشكال التنسيق و الجدولة فوق الشبكة لنضمن اتزان الحمولة و القدرة على التسلق. وأخيرا استعملنا تراكيب مثالية للشبكة لدراسة أداء الحسابات و إثارة الجدل حول فعاليتها بدون التقيد بتركيبة خاصة ترقبا لأنظمة الحاسب المستقبلي.

Abstract

The objective behind this research work is to design efficient parallel solutions to selected medical imaging algorithms in terms of performance, scalability and generality. We have selected algorithms which present a considerable complexity and heavy load and designed solutions based on one hand, parallel programming *concepts to exploit the parallelism inherent in the algorithm*. On the other hand, encoded medical data sets with encoding schemes to enforce a structure so that we can identify coherent regions and save processing time for homogenous regions. Furthermore, we have used parallel models to exploit various forms of process synchronization and scheduling over network topologies to ensure load balancing and the ability to scale up. Finally, abstract machine architectures have been used to study the algorithms and argue about their efficiency without getting too much concerned with a specific hardware.

Résumé

Ce travail de recherche consiste à la conception de solutions efficientes à des algorithmes d'imagerie médicale en termes de performance, de croissante et de généralité. Nous avons sélectionné des algorithmes qui présentent une complexité considérable et une importante charge et conçu des solutions basées sur deux concepts. Le premier consiste a exploiter le parallélisme inhérent dans l'algorithme tandis que le deuxième consiste a coder les donnés avec une structure qui nous permet d'identifier les régions cohérentes et par conséquent réduire le temps d'exécution pour les régions homogènes. En outre, nous avons utilisé les modèles parallèles pour exploiter une variété de formes de synchronisation et de planification sur les topologies de réseaux pour assurer la balance de charge et l'habilité de croissance. Finalement, nous avons utilisé des architectures de machine abstraite pour étudier les algorithmes et de débattre leurs efficacités sans se soucier d'un hardware spécifique.

Table of contents

1 Introduction 1

1.1 Research statement 2
1.1.1 Motivation 2
1.1.2 Research objectives 3
1.1.3 Research approach 4
1.2 Identification of medical imaging algorithms and case studies ……..... 5
1.2.1 Information flow in a medical imaging system ……..... 5
1.2.2 Data acquisition 5
1.2.3 Image processing 6
1.2.4 Model creation 7
1.2.5 Viewing operations 7
1.2.6 Generation of realistic images 7
1.3 Concepts used in the design of the parallel algorithms 9
1.3.1 Parallel models used to exploit the parallelism inherent in the algorithm …… 9
1.3.2 Data encoding used to exploit data parallelism .....…… 9
1.3.3 Synchronisation through messages (parallel models (MPS & BSP) .. 9
1.3.4 Distributed memory systems .....… 10
1.3.5 Mating other models to the algorithms 10
1.4 Evaluation of performance .....….. 11
1.5 Thesis organisation .....…. 12

2 Medical imaging (Algorithms, software packages, computational requirements, medical imaging applications) ……. 13

2.1 Medical imaging algorithms .....… 14
2.1.1 Acquisition algorithms and their computational requirement ……..... 16
2.1.1.1 Sampling and reconstruction algorithms 16
2.1.1.2 Reconstruction algorithms 18
2.1.1.3 Computational requirements 19
2.1.2 Resampling and interpolation algorithms 20
2.1.3 Representation of 3D medical objects and encoding algorithms..…..... 21
2.1.3.1 Boundary representations 24
2.1.3.2 Volume representations 24
2.1.3.2.1 The VDE file format 26
2.1.3.3 Octrees and Quadtrees 27
2.1.3.3.1 Octree and Quadtree data structures 29
2.1.4 Segmentation algorithms 31
2.1.4.1 Image segmentation 31
2.1.4.2 Edge detection algorithms 33
2.1.4.3 The elastic matching algorithm (mssm) 34
2.1.5 Visualisation algorithms .....… 36

2.1.5.1 Volume rendering 36
2.1.5.2 Octree visualisation method 40
2.2 Medical imaging systems, software tools and their computational requirements 42
2.2.1 Medical imaging computational requirements 42
2.2.2 Approaches to the use of 3D graphics in medical imaging 42
2.2.3 Medical graphics systems 44
2.2.3.1 ANALYSE software package 44
2.2.3.2 The VDE software 46
3 Parallel computers , performance measures, models of parallel computations 48

3.1 Parallel computers 49
3.1.1 Introduction to parallel computers and Distributed Memory Multicomputers 49
3.1.1.1 Pipeline computers 49
3.1.1.2 Vector computers 50
3.1.1.3 Multiprocessors 51
3.1.1.4 Distributed Memory Multicomputers 55
3.1.2 Transputer and related systems 58
3.1.2.1 Transputer architecture & support of concurrency 58
3.1.2.2 Support of concurrency 59
3.1.2.3 The INMOS link 61
3.1.2.4 Performance 63
3.1.2.5 Building systems with transputers 64
3.2 Performance measures 65
3.2.1 Introduction (Amdahl's law and other laws) 66
3.2.2 Definitions and laws 67
3.2.2.1 Crosch's law 67
3.2.2.2 Performance and technology 68
3.2.2.3 Von Neumann's Bottleneck 68
3.2.2.4 Amdahl's law 69
3.2.2.5 The Gustaffson-Barsis law 72
3.2.2.6 Fine grain vs coarse grain machines and R/C ratio 73
3.2.3 Performance models 75
3.2.3.1 Basic model: Two- processors with unoverlapped communications 75
3.2.3.2 Extension to N processors 77
3.3 Models of parallel computation 81
3.3.1 Introduction 81
3.3.2 Techniques for exploiting parallelism 82
3.3.2.1 The processor farm 82
3.3.2.2 Algorithmic parallelism 82
3.3.2.3 Geometric parallelism 83
3.4 Parallel languages 84
3.4.1 Introduction 84
3.4.2 The ANSI C toolset 84
3.4.2.1 Introduction 84
3.4.2.2 Support for earlier tools 85
3.4.2.3 ANSI C toolset 85
3.4.2.3.1 ANSI C compiler 85

3.4.2.3.2 Generating executable code 85
3.4.2.3.3 Loading and running programs 85
3.4.2.3.4 Program development support 86
3.4.2.3.5 Runtime library 86
3.2.2.3.6 Environment variables 87
3.4.3 Parallel processing with ANSI C 87
3.4.3.1 Process, Channel, and Semaphore data types 87
3.4.3.2 Concurrency functions 88
3.4.3.3 Processes 88
3.4.3.4 Channel communication 92
3.4.3.5 Parallel programming examples 94-97
3.5 Parallel programming using ANSI C toolset on MIMD networks supported by the IMS B008 transputer board and The S708 software support 98
3.5.1 Introduction to the IMS B008 Board 99
3.5.1.1 The IMS B008 Transputer board 99
3.5.1.2 Configuration through switches 99
3.5.2 The MMS Module Motherboard Software (S708 device driver) 100
3.5.2.1 Installing the S708 device driver 100
3.5.2.2 Using the MMS to run application programs 100
3.5.2.3 Menu options 100
3.5.2.4 Description of the softwire configuration 100
3.5.2.5 Network mapper 101
3.5.3 Installing the IMS D7214 ANSI C toolset 101
3.5.3.1 Introduction 101
3.5.3.2 Installing the release 102
3.5.3.2.1 Installation 102
3.5.3.2.2 Setting up the toolset for use 102
3.5.3.3 confidence testing 103

4 Design of two parallel solutions to the mssm algorithm with a logically fully connected network supporting message passing 104

4.1 Parallel solution 1: (Task decomposition & Pipelined model) 105
4.1.1The model 105
4.1.2 Programming with Cstools 107
4.1.3 Performance analysis 109
4.1.4 Conclusion 112
4.2 Parallel solution 2: (Data partitioning & Process farm) 113
4.2.1 The process farm model 114
4.2.2 Data partitioning and task granularity 115
4.2.3 Matching windows and data requirements 117
4.2.4 Data replication and locality of data access 117
4.2.5 Task scheduling (demand driven) 119
4.2.6 Synchronisation using buffered messages 119
4.2.7 Programming parallel solution 2 with Cstools 121
4.2.8 Performance of the parallel solution 2 122
4.3 Conclusion 124

5 Mating LINDA, BSP and SM parallel programming models with the MSSM algorithm over distributed memory MIMD networks 127

5.1 Introduction 128
5.2 Designing with Linda parallel programming model
(Uncoupled programming) 129
5.2.1 The model 129
5.2.2 Uncoupled programming (The interface) 130
5.2.3 Programming with C-Linda the elastic matching problem 130
5.2.4 Performance of the model 132
5.3 Shared objects (abstract data types) 132
5.3.1 The model 132
5.3.1.1 Shared Objects 133
5.3.2 The interface 134
5.3.2.1 Data structures 134
5.3.3 Programming the elastic matching problem with SO model 135
5.3.4 Performance of the model 137
5.4 XPRAM programming model 137
5.4.1 The Underlying Machine Model 137
5.4.2 The XPRAM programming model 138
5.4.2.1 Process management 139
5.4.2.2 Memory access 139
5.4.2.3 Shared space 140
5.4.2.4 Process synchronisation 140
5.4.3 A programming interface for the XPRAM model 141
5.4.3.1 Process management 141
5.4.3.2 Shared memory 141
5.4.3.3 Process synchronisation 141
5.4.3.3.1 Futures 141
5.4.3.3.2 Bulk synchronous operation 142
5.4.4 Programming with XPRAM model 142
5.4.5 Performance of the model 145

6 Parallelization of the Octree visualisation algorithm 146

6.1 Introduction 147
6.2 The algorithm 148
6.2.1 The octree visualisation algorithm 148
6.2.2 The application software :Parallel Viewer Software (PVS)..... 150
6.3 The parallel model 151
6.4 Data granularity and task scheduling 152
6.5 Messages and their protocols 153
6.6 Process synchronization of the process farm model 154
6.6.1 Process spawning and channel links 154
6.6.1.1 CreateChannels and CreatePlist. 154
6.6.2 The farmer process 155
6.6.3 The worker process 156
6.7 Design of the functions of the Parallel Viewer Software (PVS)..... 157
6.7.1 Acquision and AdjustView..... 157
6.7.1.1 Acquisition 157
6.7.1.2 AdjustView 158
6.7.2 The octree visualization algorithm (The Vision Pipe) 159

6.7.2.1 Encoding: BuildOctree(), InitOctNode(), TestOctant() …. 159
6.7.2.2 Mapping: OctToQuad(), BuilQuadNode(), and Destroy…... 163
6.7.2.3 Display: TraverseQuadTree(), FillBuffer() 166
6.8 Implementation 168
6.8.1 The PVS (Parallel viewer software). 168
6.8.2 Global and functions prototypes (pvs.h) 173
6.8.3 Graphics library 175
6.8.4 Network library ……. 175
6.8.5 Data partitioning library: SavePgrain() ……..... 175
6.8.6 Vision library 176
6.9 Performance of the parallel solution .....…. 176
6.10 Conclusion .....…… 177

7 General conclusion ……...…… 178-179

Templates …… 180-185

Template 1:The interface window of the VDE software ……. 180
Template 2:2D scanned slices of a head (Kennedy87] …….. 181
Template 3: Low pass filtering (slice number 3 of template2) ……... 182
Template 4: High pass filtering (slice number 3 of template 2) ……. 183
Template 5: Visualization of the head …… 184
Template 6: Interface window of the PVS shows the software running with 4 processors 185

References .....…190-198

Chapter 1:

Introduction

Contents:

1.1 Research statementMotivation
1.1.1 MotivationJustification
1.1.2 Research oObjectivesApproach
1.1.3 Research aApproachpplications
1.1.4 Design tools
1.1.5 Implmentations
1.2 Identification ofntoduction to medical imaging algorithms and case studies
1.2.1 Information flow in a medical imaging systemdentification of medical imaging algorithms
1.2.2 Selection of algorithms for case studies
1.2.1 Identification of medical imaging algorithms
1.2.21.1 Data acquisition (2D in BMP format and 3D in Octree format)
1.2.1.32 Image processing (Blurring, segmentation, matchingm "mssm")
1.22.1.43 Model creation (Image space)
1.2.1.54 Viewing operations (Perpective view, Internal structure)
1.2.1.65 Generation of realistic imagesDisplay (Ray-casting)

1.2.2 Medical imaging systems, software packages and their computational requirements
1.2.2.1 Compact and efficient software packages designed to run on the hardware of the data acquisition device.
1.2.2.2 Portable software packages designed to run on the stand-alone dedicated computing platforms.
1.2.2.3 Software packages running on a special purpose hardware
1.2.2.4 Concurrent software running on concurrent supercomputers
1.2.2.5 Interactive software running on super minicomputer
1.3 CDesign & implementation "Application of parallel processing to medical imagingoncepts used in the design of the parallel algorithms"
1.3.1 Parallel models used to exploit the parallelism inherent in the algorithms
1.3.2 Data encoding used to exploit data parallelism
1.3.3 Synchronization through messages (parallel models (MPS & BSP)
1.3.4 Distributed memory systems
1.3.5 Mating other models to the algorithms
1.3.1 1 Design toolsThe approach (Design // solutions for a wide range of // machines through a case study: the mssm algorithm)
3-Parallel machines and environment (MIMD:Attached processors with Cstools, and ANSIC).
1-Parallel models (MPS, BSP,...)

Compilers (OCCAM, ANSIC, ...)
Simulations (SM and BSP)
Applied concepts in the design of the parallel solutions (process farm , process decomposition, data partitioning)
2-Medical data sets encoding (2D:readbmp(), 3D:buildoctree())
1.43.2 Evaluation Evaluaion of performance
Image processing (Blurring and matching)
display (encoding and visualization)
Mating parallel models Design tools
1.3.3 Evaluation of performance
1.3.4 The applied concepts in the design of // solutions to the mssm algorithm
1.54 Thesis organization

1.1 Research statement

1.1.1 Motivation

Parallel computers provide an opportunity for increased performance in many problem domains. Potential benefits arise both from faster execution speed, resulting from many processes working on the same problem, and from larger problem size, as parallel systems typically have more physical memory than serial systems. However, the use of parallel computers in the field of medical imaging systems has not been based on a unified architecture and has largely been limited to research laboratories and universities. Industry has been slower to acquire technology for every day use.

The computational requirements needed to process the algorithms involved in the different stages of a typical medical imaging system (refer to figure 1.1) are considerably high. These algorithms process huge amount of data sets which represent either the 2D slices or 3D volumes of encoded data files of scanned organs obtained from various medical imaging modalities [Cho93] such as the Computerized tomography (CT) or magnetic resonance imaging (MRI).

In this modest work, we want to present the following three main topics. One: the design of parallel solutions to some algorithms involved in the various stages of a medical imaging system.Two: the selection of a suitable encoding scheme of the data sets to be able to apply spacial parallelism as well as saving space. Three: mat parallel models to the algorithms to be

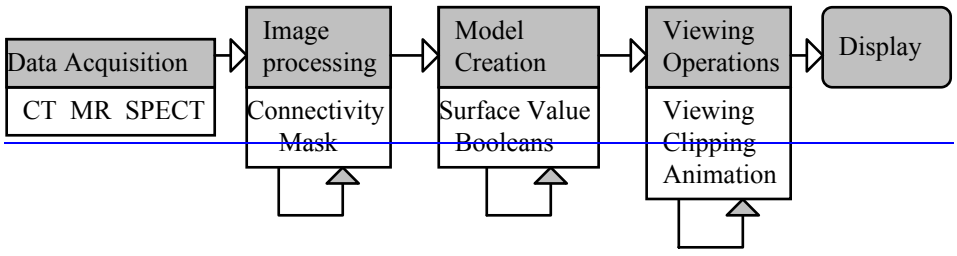


Figure 1.1: Information Flow in medical imaging systems.

able to generalize the study and consider a large family of computer systems.

1.1.2 Research objectives

Our research objective concerns the application of parallel processing to the algorithms belonging to the various stages of a typical medical imaging system (see figure 1.1). This work is faced with the fact that medical imaging systems are not based on a unified architecture as you can see later in chapter 2. Therefore we want to demonstrate that we can achieve the design of efficient parallel solutions to MI applications [Bajesy89, Beaulieu89, Borges84, Burt81, Cline88, Freider85, Kennedy87, Lenz86, Levoy88, Lifshitz90, Mowforth89, Udupa82, Yves93]. The efficiency we seek is measured in terms of performance, scalability and generality in such a way that they will reach high performance and efficiency over a large number of parallel machines. Therefore, we contribute in the generalization of the parallel programming paradigm in medical imaging applications over medical imaging systems. Consequently we can port our parallel software to a large family of computer systems including the encoming systems which will be based on new technologies. Moreover, we can ensure the scalability of our parallel software to support the growing application demands with minimum cost (i.e. scaling up system resources).Studying the requirements of medical imaging systems and of their medical applications reviel two important aspects. The first aspect is related to the computational complexity and heavy load of the algorithms involved in various imaging tasks. The second aspect is related to the real time and throughput requirements of most medical applications (i.e. quantification, visualisation, etc...).

The The contemporary field of medical imaging (MI) utilize a variety of imaging modalities such as the computerized tomography (CT) and the magnetic resonance imaging (MRI). These imaging modalities [Cho93] scan the human organs and produce huge amount of data formattedformated in 2D slices or 3D volumes of data sets. This data is processed when it goes through the data flow of a typical medical imaging system (refer to fig1.1)

~~according to the diagnosis procedure fixed by the physician in MI applications~~). For more precision~~precisions~~ we mention~~memsion~~ the three dimensional reconstruction of tomograms [Cline88], the three dimensional display from cross-section tomographics images [Kennedy87] and the model based tissue differentiation in MR brain images [Mowforth89].

Processing medical information is very computationally demanding and require powerful computer systems. This fact has led the researcher to build medical imaging systems not based on a unified architectures. These efforts have been done to devise solutions to the requirements of medical applications such as speed and throughput. The architecture of these medical imaging systems ~~have been~~ varied~~y~~ from compact and efficient software packages designed to run on the hardware of the data acquisition device such as the 3D98 [Udupa87] to p~~P~~ortable software packages designed to run on the stand-alone dedicated computing platforms such as the ANALYSE at MAYO clinic, biodynamics research unit, Minnesota~~minnesota~~ [Robb89~~xx~~] to ~~—~~concurrent software running on concurrent supercomputers such as the BODYSCAN [Mills90] running on the Edinburgh~~edinburgh~~ parallel computing centre~~center~~ (EPCC). An overview of some existing medical imaging systems is presented in chapter 2.

Studying the requirements of medical imaging systems and of their medical applications reveal two important aspects. The first aspect is related to the computational complexity and heavy load of the algorithms involved in various imaging tasks. The second aspect is related to the real time and throughput requirements of most medical applications (i.e. quantification, visualisation, etc...).

~~Parallel computers provide an opportunity for increased performance in many problem domains. Potential benefits arise both from faster execution speed, resulting from many processes working on the same problem, and from larger problem size, as parallel systems typically have more physical memory than serial systems. However, the use of parallel~~

~~computers in the field of medical imaging systems has not been based on a unified architecture and has largely been limited to research laboratories and universities. Industry has been slower to acquire technology for every day use.~~

A considerable time is associated with medical imaging algorithms~~diagnosis~~. This is due to the complexity of the algorithms and the huge amount of data upon which they operate. Some examples are given here to exemplify these constraints. -The visualization of medical data sets [Levoy88] or the extraction~~extration~~ of tissues of interest [Bajcsy89] involve very complex algorithms such as three dimensional viewing of organs [Hearn96] and matching between cuts of organs [Jain89]. The three dimensional viewing of organs implemented by the ray-tracing techniques [Hearn96] takes ours to generate the view when using a sequential processor. The algorithm compute the intersection of all the rays emanating from the viewing plan with all the voxels in the viewing direction. As a final example and not last that we state here t~~T~~he elastic matching between cuts of organs [Jain89, Bouklachi92] takes minutes to compute the discrepancy maps necessary for tissue extraction~~extractions~~ or the identification of local diseases~~deseases~~. The algorithm performs a pipeline of a 2D correlation~~corrolation~~ operator over the image space beside other computations.

1.1.2 Research objectives

The ~~two main~~ objectives of this ~~e~~ work is~~are~~ to explore approaches to parallel program design and investigate the potential of parallel implementations~~implemetations~~ for algorithms used to process and visualize medical data sets. Furthermore, use the adequate encoding techniques suitable for the representation of medical data sets.~~of visualization and elastic matching algorithms.~~ On one hand, we should invest considerable computational resources and use the most sophisticated~~sophysticated~~ computational techniques to devise solutions to perform the diagnosis process in minimum time. This will be achieved by exploiting the parallelism inherent in the applications and increase the performance of the algorithms

provided by parallel computers. On the other hand, ~~the~~ we want to devise affordable solutions which can be implemented~~implementated~~ with reasonable~~resonable~~ cost. Like having the diagnosis process being~~beeing~~ partly accomplished in the physician cabinet over an IBM PC or over a network service. In this thesis we want to contribute to solve the dilemma~~delema~~ between the diagnosis requirements and devising affordable medical imaging solutions. ~~We exploit the parallelism inherent in the applications and increase the performance of the algorithms provided by parallel computers.~~

~~Therefore we want to demonstrate that we can achieve the design of efficient parallel solutions to two MI applications. The first: the elastic matching [Bajcsy89, Lifshitz90, Mowforth89, Yves, Bouklachi92,Yves93]. The second: Three dimensional visualization of human organs [Cline88, Kennedy87,Lenz86, Udupa82, Bouklachi 97].~~

The efficiency we seek is measured in terms of performance, scalability and generality in such a way that they will reach high performance and efficiency over a large number of parallel machines. Therefore, we want to contribute in the generalization of the parallel programming paradigm in medical imaging applications over medical imaging systems. Consequently we can port our parallel software to a large family of computer systems including the i~~e~~ncoming systems which will be based on new technologies. Moreover, we can ensure the scalability of our parallel software to support the growing application demands with minimum cost (i.e. scaling up system resources).

~~We present in chapter 4 a work to mat parallel programming models the elastic matching algorithm [Bouklachi 97]. This work shows the generalization of the parallell programming paradime through conciving solutions to the mssm algortithm when using parallell programming models such as the MPS and BSP and others.~~

~~1.1.3~~1.1.34 Research approach

This is followed through the presentation of solutions at three levels:

One: The design of parallel solutions to selected algorithms which present a considerable complexity and heavy load. These solutions are designed based on parallel programming concepts, models and abstract machine architectures over MIMD networks. The programming concepts are used *to exploit the parallelism inherent in the algorithm or in the data*. While the parallel models are used *to exploit various forms of process synchronisation~~synchronizations~~ and scheduling~~sheduling~~ over network topologies. Finally the MIMD networks are selected due to their ability to scale up* and to promote to general purpose parallel computers [Valiant88] if we view them as abstract machine architecture (i.e. fully connected networks).

Two: The selection and the design of ~~a~~ suitable encoding schemes [Samet88, book94, Mougari99] to encode the medical data sets. The encoding scheme enforce a structure so that we can identify coherent region and save processing time for homogenous regions. Furthermore, this spatial~~spacial~~ enumeration enables us to partition the data and exploit fine to coarse granularity and distribute the load over the network processors for a better load balancing.

Three: mat parallel models to the algorithms to be able to generalize the study and consider a large family of computer systems [Bouklachi97]~~— Our approach to achieve this goal is based on one hand, on programming~~. We will program our MI applications with advanced parallel programming models and abstract machine architectures [Harris94]. This will allow us the study of the algorithm and argue about its efficiency without getting too much concerned with a specific hardware. We present in chapter 4 a work to mat parallel programming models the elastic matching algorithm [Bouklachi 97]. This work shows the generalization of

the parallel programming paradigm through conceiving solutions to the mssm algorithm when using parallel programming models such as the MPS and BSP and others.

~~On the other hand, by using adequate programming concepts such as data partitioning to *exploit the granularity of the data*.~~

~~The design of the parallel solutions is based on parallel programming concepts, parallel programming models and abstract machine architectures over MIMD networks. The programming concepts are used *to exploit the parallelism inherent in the algorithm or in the data*. While the parallel models are used *to exploit various forms of process synchronizations and sheduling over network topologies. Finally the MIMD networks are selected due to their ability to scale up* and to promote to general purpose parallel computers [Valiant88] if we view them as abstract machine architecture (i.e. fully connected networks).~~

~~1~~1.2~~1.4~~ Identification of medical imaging algorithms and case studies

1.2.1 Information flow in medical imaging systems

The information flow in a typical medical imaging system can be decomposed into five stages as shown in figure 1.1. These stages are data acquisition, image processing, model creation, viewing operations and finally display (refer to chapter 2 for more detail).

1.2.2 Data acquisition:

The data acquisition concern the interaction of all forms of radiation with tissue and the development of appropriate technology to extract clinically useful information from observations of this interaction. Such information is usually displayed in an image format (2D slices). Medical images can be as simple as projection or shadow image -as first produced by Roentgen in 1895 nearly 100 years ago and utilized today as a simple chest X-ray -or as complicated as a computer reconstructed image -as produced by computerized tomography

(CT) using X-rays or by magnetic resonance imaging (MRI) using intense magnetic fields.~~**1.2.1 Identification of algorithms:**~~

~~Application~~

~~The information flow in a typical medical imaging system can be decomposed into five stages as shown in figure 1.1. These stages are data acquisition, image processing, model~~

.

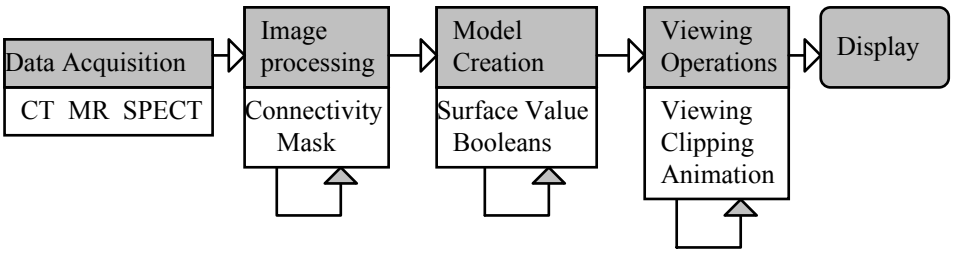


Figure 1.1: Information Flow in medical imaging systems.

~~creation, viewing operations and finally display (refer to chapter 2 for more detail).~~

~~**1.2.2 Data acquisition:**~~

~~The data acquisition concern the interaction of all forms of radiation with tissue and the development of appropriate technology to extract clinically useful information from observations of this interaction. Such information is usually displayed in an image format (2D slices). Medical images can be as simple as projection or shadow image -as first produced by Roentgen~~Rontgen ~~in 1895 nearly 100 years ago and utilized today as a simple chest X-ray -or as complicated as a computer reconstructed image -as produced by computerized tomography (CT) using X-rays or by magnetic resonance imaging (MRI) using intense magnetic fields.~~

This is the first step is performed by the medical imaging hardware; It samples some property in a patient and produces multiple 2D slices of information. The data sampled depends on the data acquisition technique. 2-D and 3D

1.2.3 Image processing

The Image processing has grown remarkably since its introduction in 1972. There has been a rise in the utilization of various aspects of image processing in many fields, often

coupled with recent developments in digital computers and related signal processing technologies. Image processing now plays an important role in medicine; it can be used by physicians, medical engineers, and physicists to educe maximal information from detected images and to diagnose diseases~~deseases~~. Some algorithms use image processing techniques to find structures within the 3D data or to remove artefacts~~artifacts~~ and filter the data if it is noisy. Others use interpolation and reconstruction for viewing the final image.

An algorithm called the Multiple Scale Signal Matching or in short the *MSSM* [Mowforth89, Bajcsy89] is taken as a case study in this thesis. The MSSM algorithm is applied in medical diagnosis to compute discrepancy maps to match anatomical models or to extract from them tissues of interest. The algorithm is based on segmentation and matching. The segmentation detects lines and edges in the image according to Marr's conjecture [Marr82] concerning the representation of visual information. The matching is performed elastically due to the nature of the human tissues which do not have regular shapes[Bajcsy89, Borges84].following Marr's conjecture [Mars82].

1.2.4 Model creation

The model creation involves the creation of surface model from the 3D data. The model usually consists of 3D volume elements (voxels) or polygons~~polygones~~. Users select the desired surface by specifying a density value. This step can also include the creation of cut or capped surfaces. The 3D objects can be an octree representations of 3D arrays.

1.2.5 Viewing operation

The viewing operations for modelling~~modeling~~ and converting a world-coordinate description of a scene to device coordinates are called the viewing pipeline[Hearn94]. Once the scene has been modelled~~modeled~~, world-coordinate positions are converted to viewing coordinates. The viewing -coordinate system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane, which we

can think of in analogy with the camera film plane. Next, projection operations are performed to convert the viewing coordinate description~~discription~~ of the scene to coordinate positions on the projection plane, which will then be mapped to output device. Objects outside the specified viewing limits are clipped from further considerations, and the remaining objects are processed through visible-surface identification and surface-rendering procedures to produce the display within the device viewport.

1.2.6 Generation of realistic images

The generation of a realistic graphics displays has to identify those parts of a scene that are visible from a chosen~~choosen~~ viewing position. There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Deciding upon a method for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. the various algorithms are referred to as *visible-surface elimination methods*. Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images. These two approaches are called *object-space methods* and *image-space methods*, respectively. An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is detected point by point at each pixel position on the projection plane.

The visible-surface detection algorithms follow many methods to solve the visibility problem. Some of these algorithms use the Back-To-Front (BTF) methods [Freider85], octree method [Meagher82] ,and ray casting method [Hearn96].

The *ray-casting* algorithm [Hearn96] is used to visualize the 3D data sets of scanned human organs. An image of a volume data can be calculated by casting a visibility or primary ray through each pixel in the image into the scene. The primary rays are classified against the octree using a recursive divide and conquer algorithm. This intersects the ray against each halfspace in the tree merging the results using the set operators until a valid intersection point is found. This point along with a vector representing the surface normal at he point is then passed to the illumination model. The illumination model uses lighting information and surface characteristics, such as colour and texture , to calculate an illumination value for the pixel the primary ray started from. The process of generating an image by tracing a single ray per pixel is usually referred to as ray casting. Roth [Roth86] and Hearn [Hearn94] provide a detailed description of ray casting algorithm.

1.2.2 In this thesis we have selected from a wide class of medical imaging algorithms some representatives of medical imaging algorithms. The first algorithm is the *ray-casting* algorithm [Hearn94] to visualize the 3D data sets of scanned human organs.The second algorithm is called the Multiple Scale Signal Matching or in short the *MSSM* [Mowforth89, Bajcsy89]. These algorithms are characterized by their computational complexity and heavy load. The MSSM algorithm is applied in medical diagnosis to compute descripancy maps to match anatomical models or to extract from them tissues of interest. The algorithm is based on segmentation and matching. The segmentation detects lines and edges in the image according to Marr's conjecture [Marr82] concerning the representation of visual information. The matching is performed elastically due to the nature of the human tissues which do not have regular shapes[Bajcsy89, Borges84].

Case studies:1.1.5 Strategy

We have selected the ray-casting and the multiple scale signal matching algorithms for their computational complexity and heay load.In our work strategy we have selected We will describe them in short in the present paragraph.,

in one hand, The fhe *ray-casting* algorithm [Hearn96] is used to visualize the 3D data sets of scanned human organs. An image of a volume data can be calculated by casting a visibility or primary ray through each pixel in the image into the scene. The primary rays are classified against the octree using a recursive divide and conquer algorithm. This intersects the ray against each halfspace in the tree merging the results using the set operators until a valid intersection point is found. This point along with a vector representing the surface normal at he point is then passed to the illumination model. The illumination model uses lighting information and surface characteristics, such as colour and texture , to calculate an

illumination value for the pixel the primary ray started from. The process of generating an image by tracing a single ray per pixel is usually referred to as ray casting. Roth [Roth86] and Hearn [Hearn94] provide a detailed description of ray casting algorithm.

The second algorithm is called the Multiple Scale Signal Matching or in short the *MSSM* [Mowforth89, Bajcsy89]. The MSSM algorithm is applied in medical diagnosis to compute descripancy maps to match anatomical models or to extract from them tissues of interest. The algorithm is based on segmentation and matching. The segmentation detects lines and edges in the image according to Marr's conjecture [Marr82] concerning the representation of visual information. The matching is performed elastically due to the nature of the human tissues which do not have regular shapes[Bajcsy89, Borges84].following Marr's conjecture [Mars82]. a representative of medical imaging algorithms which is the MSSM.

On the other hand, we have selected a variety of abstract machine architectures and parallel programming models to design a parallel algorithm which high efficiency and performance characteristics. At first, we have selected a parallel programming environment which is based on a message passing system (MPS) which views a transputer based parallel machine (MIMD) as a fully connected abstract machine and which support message routing. In subsequent parallel solutions we have selected a parallel model based on BSP and an abstract machine architecture which views the MIMD network as a fully connected network which support abstract shared memory.

1.3 Concepts used in the design of the parallel algorithms~~Design & Implementation "Application of parallel processing to medical imaging"~~

We will design parallel solutions to ~~by the parallelization of~~ the octree visualization~~ray-casting~~ algorithm and to the msssm algorithm by applying the following concepts:

1.3.1 Parallel models used to exploit the parallelism inherent in the algorithms:

Task decomposition versus data partitioning strategies, pipelined versus process farm models, and finally distributed versus shared memory abstractions.

1.3.2 Data encoding used to exploit data parallelism: ~~(Medical data sets encoding(2D: readbmp().~~

The data encoding of the medical data is currently performed in many ways. The 2D slices representing cuts of human organs can be represented in bitmaps [data95] or other compressed forms such as PCX or JPEC. The 3D data sets are volumes which represent the scanned organs produced by modern medical imaging systems. The most known encoding scheme of the 3D data sets are octrees and the standard is called JP2 [octree].

The author has contributed in the development of C code to read and display 2D encoded images [Khodja&Bouklachi96]. Furthermore, he has designed a new standard called Volume Data Encoding (VDE) [Mougari&Laroui&Bouklachi 99]. The VDE format encodes the volume data in file headed with headers, color table, and followed by a set of slices representing the cuts of the human organ (refer to section 2.1.3.2.1)~~n~~.

Partitioning the data sets into regular regions of various granularity~~ies~~ (fine grain to large grain). This is performed by partitioning the image slice into a grid in the x and y direction and considering a granularity G (refer to chapter 4). For the volumetric data we will partition the volumetric data according to a concept called orthogonal tunnel (refer to chapter 6)~~the octree structure (eight octants)~~. The implementation details are explained in the corresponding chapters.

~~*The original 2D slices of the human head obtained from Leeds university have been encoded in SUN format. The hauthor has designed software to convert the 2D SUN format to BMP format to be able to decompress and process the encoded data using the ANSIC compiler under DOS and IBMPC.*~~

~~*The volumetric data (3D) representing the organs have not been available in Algeria. Therefore, the hauthor have designed software to model and encod the 3D volume data in an Octree formats [Bouklachi97].*~~

1.3.3~~2~~ Synchronization through messages: (Parallel models(MPS, BSP)

We have selected one a class of ~~a~~ parallel programming environments which are~~is~~ based on ~~a~~ message passing system (MPS). These environments are the parallel C language compilers under UNIX~~unix~~ and ANSI C under DOS~~MSDOS~~. These MPS systems views a

transputer based parallel machine (MIMD) as a fully connected abstract machine and which support message routing. In subsequent parallel solutions we have selected a parallel model based on BSP and an abstract machine architecture which views the MIMD network as a fully connected network which support abstract shared memory.

Two: Parallel programming environments such as Visual C++ under windows95, XPRAM BSP based model, and many other hybrid models. These models were used to simulate many aspects of the parallel program design and to develop code. For instance the octree volume visualization code has been experimented in visual C++ under windows95 first. This code has given rise to many discussions and ideas for the implementation of parallel code with ANSIC over the transputer target machines.

1.3.4~~3~~ Distributed memory systems:

The parallel machines over which we have designed our parallel software are attached transputer networks. These machines are classified as MIMD machines due to their architecture. The first parallel machine is a network of transputers attached to a meiko computing platform running Cstools parallel programming environment. The second parallel machine is a network of transputers attached an IBMPC running ANSIC toolset under DOS environment. The parallel machines have been attached processors to host computers. The first is a set of transputer boards attached to a VAX~~vax~~ computer, the second parallel machine is a set on B004 boards attached to an IBMPC (refer to chapter 3~~define the chapter and the section where the presentation of the B004 board is to be adequate~~).

1.3.5 ~~With these parameters set up (algorithm, parallel models and abstract machine architectures) we have designed machine independent software. This software is represented in a set of parallel solutions to the mssm algorithm. In these solutions we have experimented task decomposition versus data partitioning strategies, pipelined versus process farm models and distributed versus shared memory abstractions. We~~

~~have concluded with a performance evaluations of each method in terms of speed of execution, scalability and portability (generality).~~

~~1.1.5 Implementation~~

~~The implementations of the parallel solutions have been carried on different parallel machines and environments. The first solution have been programmed on a meiko computing platform based on a network on transputers and Cstools parallel programming environment. The second~~**Matting other models to the algorithms**

Some of the work has been carried under ~~two other~~ hybrid~~concurrent~~ programming models and environments~~environments~~ to achieve many objectives discussed in this thesis. Among these models ~~The first~~ is a~~s~~ ~~solution is based on a~~ simulator based on the XPRAM model running under UNIX and SUN workstations. Another~~.~~ ~~The second~~ is the preemptive~~multiprocessing~~ model under windows95 environment using visual C++ ~~run~~generating code for ~~n and~~ IBMPC. ~~Finally we have concluded with simulations on a parallel machine based on a network of Inmos Boards and hosts of PC's and ANSIC toolset. under some programming environments such as Cstools[] and Windows[]. The ray casting algorithm is programmed on windows and a PC.~~

1.2 Introduction to medical imaging

Since the introduction of computerized tomography(CT) over 20 years ago, medical imaging and general disciplines of imaging science and imaging technology have grown at a remarkable pace. In fact, the introduction of CT created the contemporary field of medical imaging, transforming classical two-dimensional qualitative imaging into a quantitative three-dimensional format. Although the CT concepts was first applied to X-rays, it has since been utilized with a host of interaction parameters leading to a variety of imaging modalities [Zang93].

1.2.1 Identification of medical imaging algorithms

The information flow in a typical medical imaging system can be decomposed into five stages as shown in figure 1.1. These stages are data acquisition, image processing, model creation, viewing operations and finally display (refer to chapter 2 for more detail). The mssm algorithm belongs to the image processing stage. It comprises a segmentation process and a matching process. These processes generate data for other image processing and visualization processes.

1.2.1.1 Data acquisition.

In medical imaging we are concerned with the interaction of all forms of radiation with tissue and the development of appropriate technology to extract clinically useful information from observations of this interaction. Such information is usually displayed in an image format (2D slices). Medical images can be as simple as projection or shadow image -as first produced by Rontgen in 1895 nearly 100 years ago and utilized today as a simple chest X-ray -or as complicated as a computer reconstructed

~~image -as produced by computerized tomography (CT) using X-rays or by magnetic resonance imaging (MRI) using intense magnetic fields.~~

~~This is the first step performed by the medical imaging hardware; It samples some property in a patient and produces multiple 2D slices of information. The data sampled depends on the data acquisition technique. 2-D and 3D~~

~~1.2.1.2 Image processing:~~

~~Since its introduction in 1972, digital image processing has grown remarkably. There has been a rise in the utilization of various aspects of image processing in many fields, often coupled with recent developments in digital computers and related signal processing technologies. Image processing now plays an important role in medicine; it can be used by physicians, medical engineers, and physicists to educe maximal information from detected images and to diagnose deseases.~~

~~Some algorithms use image processing techniques to find structures within the 3D data or to remove artifacts and filter the data if it is noisy. Others use interpolation and reconstruction for viewing the final image.~~

~~1.2.1.3 Model creation~~

~~Surface construction involves the creation of surface model from the 3D data. The model usually consists of 3D volume elements (voxels) or polygones. Users select the desired surface by specifying a density value. This step can also include the creation of cut or capped surfaces. The 3D objects can be an octree representations of 3D arrays.~~

~~1.2.1.4 Viewing operations~~

~~The viewing operations for modeling and converting a world-coordinate description of a scene to device coordinates are called the viewing pipeline[Hearn94]. Once the scene has been modeled, world-coordinate positions are converted to viewing coordinates. The viewing -coordinate system is used in graphics packages as a reference~~

~~for specifying the observer viewing position and the position of the projection plane, which we can think of in analogy with the camera film plane. Next, projection operations are performed to convert the viewing coordinate discription of the scene to coordinate positions on the projection plane, which will then be mapped to output device. Objects outside the specified viewing limits are clipped from further considerations, and the remaining objects are processed through visible-surface identification and surface-rendering procedures to produce the display within the device viewport.~~

~~1.2.1.5 Display~~

~~The generation of a realistic graphics displays has to identify those parts of a scene that are visible from a choosen viewing position. There are many approaches we can take to solve this problem, and numerous algorithms have been devised for efficient identification of visible objects for different types of applications. Some methods require more memory, some involve more processing time, and some apply only to special types of objects. Deciding upon a method for a particular application can depend on such factors as the complexity of the scene, type of objects to be displayed, available equipment, and whether static or animated displays are to be generated. the various algorithms are referred to as *visible-surface elimination methods*.~~

~~Visible-surface detection algorithms are broadly classified according to whether they deal with object definitions directly or with their projected images. These two approaches are called *object-space methods* and *image-space methods*, respectively. An object-space method compares objects and parts of objects to each other within the scene definition to determine which surfaces, as a whole, we should label as visible. In an image-space algorithm, visibility is detected point by point at each pixel position on the projection plane.~~

~~The visible-surface detection algorithms follow many methods to solve the visibility problem. Some of these algorithms use the depth-buffer methods [], octree method [] ,and ray casting method [].~~

~~1.2.2 Medical imaging systems, software packages and their computational requirements~~

~~Studying the requirements of medical imaging systems and of their medical applications reviel two inportant aspects. The first aspect is related to the computational complexity and heavy load of the algorithms involved in various imaging tasks. The second aspect is related to the real time and throughput requirements of most medical applications (i.e. diagnosis, visualisation, etc...). Therefore, processing medical information is very computationally demanding and require powerful computer systems. This fact has led the researcher to build medical imaging systems with parallel architectures to achieve the required speed and throughput of medical applications. These parallel architectures vary from supercomputers to multiprocessors and multicomputers. All of these architectures have been used in the development of various medical imaging systems. To clarify this diversity in the design we have selected to discuss some existing system and list some of their aspects in the following points.~~

~~1.2.2.1 Compact and efficient software packages designed to run on the hardware of the data acquisition device.~~

~~One of these system is the 3D98 at University of Pensilvania at Philadelphia [Herman,Udupa]. The 3D98 software package is written in FORTRAN and runs on a hardware composed of an S100 Processor and a GE9800 CT scanner. Professor G.T. Herman from the Medical Image Processing Group (MIPG) has used this system in the University of pensilvania at Philadelphia and pionered in voxel databases.~~

~~1.2.2.2 Portable software packages designed to run on the stand-alone dedicated computing platforms~~

~~One of these systems is the ANALYSE marketed by CTI (Tennesse) at MAYO clinic, Biodynamics Research Unit, Minneapolis, Minnesota [ROBB]. The ANALYSE software is written in C and enables the user to display, manipulate, measure medical datasets obtained from CT, MRI, PET and ultrasound scanners. At MAYO clinic the ANALYSE runs on a network of SUN workstations (These workstations have the UNIX operating system) and uses PIXAR a special purpose hardware composed of a four channel SIMD processor for fast display of 3D datasets.~~

~~1.2.2.3 Software packages running on a special purpose hardware~~

~~The software package INSIGHT runs on a special purpose hardware based on the VOXEL PROCESSOR [2] . This medical system has been used by the MIPG research group headed by Meager at Rensalear Polytechnic Albany, N.Y.. They have pioneered in the use of Octree representations to display and manipulate 3D medical datasets.~~

~~ChapVolumes is a software package which runs on PIXAR (a four channel SIMD processor) [Levinthal84]. PIXAR is a commercial company based at san Rafael California and their 3D hardware produces high quality images in tens of seconds.~~

~~The Graphics PARCUM system is a 3D Memory based computer architecture for displaying solid models developed at North Holland [Jackel 85].~~

~~The PIXEL-PLANES is developed at University of North Carolina at Chaptel Hill (UNC) [Fuchs89]. This system is called the polygone renderer.~~

~~A special architecture called Cubic frame Buffer (CUBE) which is based on a memory and processor architecture. It is specialized in 3D Voxel based Imagery [Kaufman88].~~

~~1.2.2.4 Concurrent software running on concurrent supercomputers~~

~~Concurrent software such as the BODYSCAN which runs on the Edinburgh Parallel Computing Center (EPCC) concurrent supercomputer[3 MILLS]. BODYSCAN is designed with OCCAM and TINY communication harness and runs on the EPCC configured as an MIMD fully connected network which is composed of 400 INMOS transputers T800 with 4Mbytes each. The software enables the display of 3D datasets in the form of voxels in tens of seconds using the rendering (D3) tool on the EPCC and four Meiko graphics (MK014) boards or graphics workstation (Silicon Graphics).~~

~~1.2.2.5 Interactive software running on super minicomputer~~

~~The UCL3D is a menu driven software package for surgical planning and runs on a 32-bit supermini NORSK DATA (ND5401). This system has been developed by Doctor Arridge in the Medical Physics Department at the University College London [Arridge]. The system provide interactive facilities for the planning , simulation, and evaluation of Maxillo-Facial Surgery. The medical datasets (CT scans) are displayed using a GEMS 33 Image frame buffer.~~

~~1.3 Application of parallel procesing to medical imaging~~

~~1.3.1 The approach~~

~~We are seeking the design // algorithms for a wide range of // machines. This will be accomplished through parallel programming models and abstarct machine architectures. A case study (the mssm algorithm) will be used to demonstrate our rerults.~~

~~1.3.2 Design tools~~

~~1.3.2.1 Parallel models based on MPS~~

~~We have used the CSTOOLS and the ANSIC toolset. Both of them are message passing systems.~~

~~1.3.2.2 MIMD machines~~

~~The parallel machine over which we have designed our parallel software are transputers networks. These machines are classified as MIMD machines due to their architecture. The first // machine is a meiko platform running cstools and the second // machine is based on INMOS transputer boards running ANSIC toolset or OCCAM.~~

~~**1.3.2.3 Medical data sets**~~

~~Images are formated in BITMAP formats. These images have been obtained from NMR scanners.~~

1.4~~3.3~~ Evaluation of performance

The parallel solutions we have obtained have been evaluated in terms of performance and efficiency. The efficiency is measured according to the speedup and the performance is measured in terms of generality and scalability.

~~1.3.4 The applied concepts in the design of the parallel // solutions to the mssm algorithm~~

~~mssm algorithm :~~ ~~(Experimental work :Applied concepts than Experimental work in paper)~~

~~The~~ design of the~~a~~ parallel solutions to the ~~ray casting and mssm~~ ~~mssm~~ algorithms ha~~s~~s been carried out in two steps. The first step exploits the concepts of task decomposition with pipelining and data partitioning with farming ~~[11,12,13]~~[Kung88, Green88]. The second step exploits shared memory and bulk synchronisation. The experiments of the first step have been carried out on the Meiko Computing Surface at The Computer Studies School , Leeds University, UK. This computing surface is based on a transputer network and the CSTOOLS programming environment. In the second step, the author discusses the performance of the designed parallel solution and shows the limitations of the parallel models used in the design. Thereafter, the author argues that in order to design an efficient parallel solution over MIMD networks we have to promote the network to support shared memory. These concepts have been grasped from existing MIMD Shared Memory Parallel Computers

[19] which are based on message switching networks, higher concepts for process synchronization[20], and the support of shared memory abstraction.

Ray-casting algorithm: Finally the author presents a parallel solution to the octree visualizationray-casting elastic matching algorithm over the transputer network (B008) under the ANSIC toolset [inmos] with the idea of spatialspacial decomposition (octree encoding) and process farm model with the ideas used in many existing interfaces such as SO[ ] and BSP[ ]. These ideas are exploited with the ANSIC toolset to design a parallel solution on an attached transputers network hosted on Personal Computers (PC's) at the (IEE) Electronics and Electronics Institute of University of Boumerdes , AlgeriaINELEC, Algeria.

Many aspects of the design has been worked out using DDE under windows 3.1 and later on visual C++ under windows95 over IBMPC's. The object oriented programming (OOP) of C++ compilers and the Graphical user interface (GUI) of windows have of a great help to design the 2D and 3D graphics functions and encoding schemes. The design the protocol of communication and interprocess communication between processes has been design with ANSIC and to design the functions to deal with the fixed granularity.

1.54 Thesis organisation

This thesis is organised in six chaptersfive sections. The first chapter states the problem as how it is solved,section introduces the medical imaging systems, the research problem and the strategy followed to tackletakle it. The second presents the multiple scale signal matching algorithm medical imaging algorithms: acquisition, resampling, representation of 3D data, segmentation, and visualization. It presents a contribution with a new file format (VDE) for volumetric data encoding. Followed by medical imaging systems and their computational requirements, and finally the MAYO ANALYSE software package and approaches to the use of 3D graphics in medical imaging.. The third shows how parallel processing concepts such as task decomposition and data partitioning are exploited in the

~~design of a parallel solution to the mssm algorithm~~ presents parallel computers, performance measures, models of parallel computations, parallel languages with an emphasis on ANSIC, and finally parallel programming using ANSIC toolset on MIMD networks supported by the IMS B008 transputer board and S708 softwire support~~. Section 4 discusses the performance of the designed parallel solution and shows the limitations of the parallel models used in the design. Thereafter, the author argues that in order to design an efficient parallel solution over MIMD networks we have to promote the network to support shared memory abstractions.~~. Chapter four presents the design of two parallel solutions to the multiple scale signal matching algorithm (mssm) for 2D image slices with a logically fully connected network supporting message passing. The first solution is based on task decomposition & pipelined model, the second solution uses data partitioning & process farm model. Chapter 5 presents the mating of various parallel models to the mssm algorithm.

Chapter 6 ~~Section 5~~ presents ~~the design~~ the parallelization of the octree volume visualization algorithm and presents the software routines developed to implement the parallel solutions. Finally It concludes by a discussion of the results.~~and implementation of a parallel solution using the ANSIC toolset , INMOS B008 board , and concepts of shared memory, parallel slackness, and bulk synchronisation.~~

Chapter 2

Medical imaging
(Algorithms, software packages, computational requirements
medical imaging applications)

2 Medical imaging
2.1 Medical imaging algorithms
2.1.1 Acquisition algorithms and computational requirements
2.1.1.1 Sampling and reconstruction algorithms
2.1.1.2 Reconstruction algorithms
2.1.1.3 Computational requirements
2.1.2 Resampling and interpolation algorithms
2.1.3 Representation of 3D medical objects and encoding algorithms
2.1.3.1 Boundary representations
2.1.3.2 Volume representations
2.1.3.3 Octrees and Quadtrees
2.1.3.3.1 Data structures
2.1.4 Segmentation algorithms
2.1.4.1 Image segmentation
2.1.4.2 Edge detection algorithms
2.1.4.3 The elastic matching algorithm
2.1.5 Visualization algorithms
2.1.5.1 Volume rendering
2.1.5.2 Octree visualization method
2.2 Medical imaging systems, software tools and their computational requirements
2.2.1 Medical imaging computational requirements
2.2.2 Approaches to the use of 3D graphics in medical imaging
2.2.3 Medical graphics systems
2.2.3.1 The ANALYSE software package
2.2.3.2 The VDE software

2.1 Medical Imaging Algorithms

Medical imaging play a major role in medicine. First ,it enable us to perform necessary imaging tasks such as acquisition, reconstruction, visualization and manipulation of information related to human anatomy in various forms. Second, it is considered as an important tool in clinical diagnosis, surgical planning, and radiation therapy and other medical applications. The use of medical imaging in medicine is attributed partly to the increasing performance/cost capabilities of modern computer technology, and partly to the increasing familiarity of clinical users with the possibilities available, and the commensurate demand for more sophisticated techniques. A number of comprehensive review articles are available however, covering both the technical aspects [Udupa87, Hohne87a] and the clinical applications [Hemmy87].

A very large number of medical imaging systems are available. The term medical imaging system is used here , rather than ‘package’, ‘workstation’, or ‘software’ because of the wide variety of implementations and capabilities offered. These vary from compact and efficient software packages designed to run in the hardware of the data acquisition device [Udupa86b, Vannier83a], to portable software packages designed to run on stand-alone dedicated computing platforms [Dev85, Robb87], or to special purpose hardware [Fishman89, Goldwasser87, Lenz86, Meagher84d, Jackel85]. An overview of the computational requirements of these medical imaging systems will be given in section 2.1.2.1.

The purpose of this chapter is to survey the algorithms related to medical imaging and identify algorithms which necessitate the power of parallel processing. However, for a matter of organization we have preffered to discuss these algorithms within their medical imaging task and referring to a given class of medical imaging systems. Therefore, in this thesis, we always reffer to interactive medical imaging systems [Udupa86b, Robb87, Goldwasser87, Meagher85c] which can acquire and process three-dimensional data sets. In order to present

medical imaging data sets, algorithms, and computing requirements, a medical system must described, and this is the purpose of this chapter. The main block diagram of a medical imaging system is shown in figure 2.1. In this later we have identified some algorithms involved in each of the imaging tasks mentioned above.

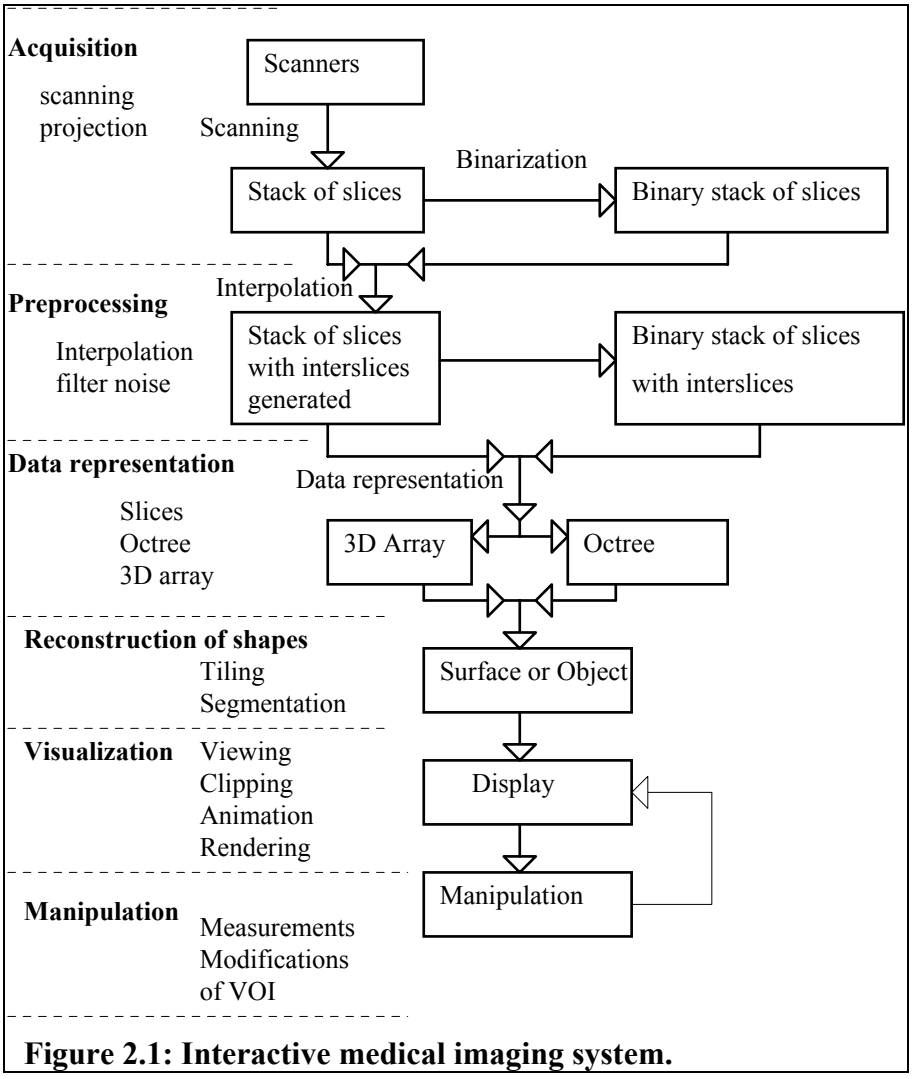


Figure 2.1: Interactive medical imaging system.

The acquisition algorithms are those which scan the medical object and reconstruct a binary stack of slices representing the scanned object. This stack of slices, in some applications, is represented with a better resolution by an interpolation algorithm or is represented with a better hierarchical data structure such as a 3D array or encoded in a Octree for random access.

Even though that a slice is a form of reconstruction, but still we need other algorithms to represent the three-dimensional shape of medical objects. This medical imaging task is the most difficult and time consuming. It involves segmentation algorithms and model creation algorithms. Many segmentation algorithms are used to reconstruct the surface of an object or to reconstruct 3D object voxel by voxel. A very important imaging task is visualization of information related to human anatomy in various forms. This task is time consuming due to the complex computations done in viewing algorithms such as rendering , clipping and animation.

In a medical surgery planning, the surgeon may manipulate the medical data by cutting some bones than want to study the effect of his surgery on the general shape of the patient. Therefore, this require selection of a Volume Of Interest (VOI), measurements, modification of medical objects and display.

In the following sections we describe in more detail these algorithms that we have mentioned. These algorithms will be discussed while we present each task of the medical imaging system which are: acquisition, interpolation (preprocessing), data encoding, reconstruction (object representation), manipulation and visualization.

2.1.1 Acquisition algorithms and computational requirements

2.1.1.1 Sampling and reconstruction algorithms

Medical images were originally limited to photographs, X-ray radiographs, or histological slides from a microscope. Therefore they were essentially two-dimensional, and recorded in an analogue fashion. Today a variety of technologies can produce two and three-dimensional data sets in machine readable fashion. This section briefly reviews the following five sources of such data and their reconstruction methods: 1) X-ray Computer Assisted Tomography (CT). 2) Nuclear Magnetic Resonance Imaging (NMR, referred to in the American literature as MRI). 3) Nuclear medicine modalities: Single Photon Emission

Computed Tomography (SPECT), Positron Emission Tomography (PET), and Time of Flight Positron Emission Tomography (TOFPET). 4) Diagnostic Ultrasound. 5) Microscopy.

Acquisition and reconstruction methods

Table 2.1: Acquisition and Reconstruction methods

Acquisition mean	Aquisition method	Reconstruction method	Reconstructed samples and the third dimension
CT	Is usually acquired by a projection obeying the 2D Radon transform	reconstructed in two dimensions , by numerically approximating the inverse Radon transform.	The reconstruction samples are 2D slices and the third dimension is acquired by moving the scanner or object along one dimension called th transaxial dimension.
NMR	Acquired by the Fourier transform of the parameter of interest.	Reconstructed by a 2D inverse Fourier transform.	samples of 2D slices
SPECT & PET	Data is obtained by projection, although attenuation and scatter limit the application of the Radon transform.	Reconstruction techniques may therefore be iterative. The projections are in 3D, but the reconstruction is usually in 2D because of math difficulties arising from incomplete data set.	2D slices and 3D distribution.
Ultra Sound	Obtained in a linear scanning mode.	The image is obtained directly.	There is no third dimension.
Micros copy	Thin slices cut from a solid object.	The image is produced directly or using modern confocal scanning techniques.	The third dimension is the set of slices.

The acquisition process is the first step performed by the medical imaging hardware. The method of acquisition may sample data in two, three, or four dimensions (the fourth dimension being time), followed by a reconstruction in two or three dimensions. It samples some property in a patient and produces multiple 2D slices of information. The data sampled depends on the data acquisition technique[FLYNN83].

1) X-ray computed tomography (CT) measures the spatially varying X-ray attenuation coefficient. The CT data is usually acquired by projection obeying the 2D Radon Transform,

and reconstructed, in two dimensions, by numerically approximating the inverse Radon Transform, CT images show internal structure.

For 3D applications, CT is frequently used to look at bone structure, although it has been used successfully in visualizing soft tissues.

2) Magnetic resonance (MR) measures three physical properties. One property is the distribution of 'mobile' hydrogen nuclei and shows overall structure within the slices. The other two properties measure relaxation times of the nuclei. MR, shows excellent constrast between a variety of soft tissues. NMR machines have a different reconstruction strategy since the data acquired is the Fourier transform of the parameter of interest, and is reconstructed by a 2D inverse Fourier Transform.

3) Single-photon emission computed tomography (SPECT) measures the emission of gamma rays. The source of these rays is a radioisotope distributed within the body. In addition to structure, SPECT can show the presence of blood in structures with a much lower dose than that required by CT. SPECT and PET data is also obtained by projection, although attenuation and scatter limit the applicability of the Radon Transform. Reconstruction techniques may therefore be iterative.

4) Ultrasound data is obtained in a linear scanning mode, and the image is produced directly. For this reason it is usually used as a real time modality.

5) In microscopy the volume data is taken either by sets of thin slices cut from a solid object, or directly using confocal scanning techniques.

2.1.1.2 Reconstruction algorithms

The reconstruction problem can be stated, in general, as a process of evaluating either the 2D (cross-section) or the 3D (volume) distribution of a physical property of an object from measured estimates of its line integrals along a finite number of paths taken through different directions. Each line integral is normally referred to as a ray integral and the collection of ray

integrals taken through a given direction is called a projection. Figure 2.2 shows one of these projections for a parallel scanning geometry in a 2D situation.

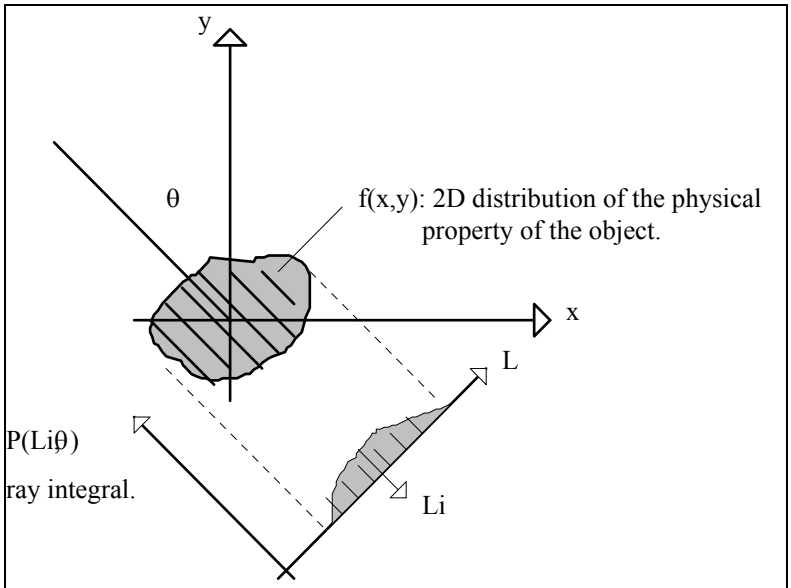


Figure 2.2: Projection at angle θ

If f(x,y) is the 2D distribution of the physical property of the object. The projection at angle θ is the ray integral shown in equation 2.1.

$$P(L_i, \theta) = \int_{L_i} f(x,y)dL_i \tag{2.1}$$

2.1.1.3 Computational requirements:

An important characteristic of tomographic reconstruction is the large amount of data that has to be processed. In a typical CT scanner, a minimum of 512 samples (ray integrals) are taken per projection and as many as 700 or more projections are obtained around the patient. The reconstructed body section is then displayed with 256 x 256 or 512 x 512 picture elements. The reconstruction time is normally in order of magnitude of the scanning time which goes up to 12 seconds. In this way, during a clinical examination, several slices can be viewed in sequence without reducing significantly the patient's throughput.

Very powerful computational resources are obviously required to accomplish this throughput. A tipical setup consist of a minicomputer which controls the gantry, handles the data acquisition and store of information and interfaces with the processor, and of an associated array of processor to perform the computations. In cases where requirements are still more stringent, solutions using special purpose processors, directly, have been proposed. Gains of about two to three orders of magnitude in the reconstruction time were reported.

The Mayo Clinic in Rochester, Minnesota, has developed a research CAT scanner for three-dimensional, stop-action, cross-action viewing of the human heart. Cross-sectional CAT images used to take 6 to 10 minutes to generate on a conventional computer. Using a dedicated array processor, the processing time can be reduced to 5 to 20 s. The image reconstruction of human anatomy in present CAT scanners is two-dimensional. It is generated too slowly (5 s) to freeze the motion of organs such as the heart or the lungs. The Mayo Clinic's super CAT scanner is expected to have 2000 to 3000 megaflops speed. It will produce three-dimensional images of beating heart, within a few seconds, with 60 to 240 thin adjacent cross sections stacked one upon the other. Because of the short processing and exposure time, three-dimensional and stop-motion pictures of a beating heart will be possible.

2.1.2 Resampling and interpolation algorithms

Resampling is used for several different purposes in image processing. An image may be resampled to a finer matrix in order to improve its appearance for image display. When an image is rotated by angles which are not a multiple of 90 degrees, resampling is required since the new coordinate points will not line up with the old points. When registering images taken with different sensorys or at different times, it may be neccessary to resample to images so that the registration is accurate to subpixel locations.

Several of these uses of image resampling are valuable for medical imaging. But, resampling for image registration and image rotation are of particular interest for digital radiology.

Several interpolation functions have been used for image resampling. The simplest is the nearest neighbor function, where the value of the new point is taken as the value of the old coordinate point which is located the nearest to the new point. Another algorithm frequently used is linear interpolation, where the new point is interpolated linearly between the old points. The next most complex functions use the four nearest points (two points in each direction). Cubic B-spline interpolating functions, which were investigated by Hou and Andrews, are positive everywhere and tend to smooth the resampled image. Cubic splines which are negative in interval (1,2) tend to preserve the original image resolution. The choice of an interpolating function to be used for resampling depends upon the task being performed.[Parker83].

Interpolation is necessary for 3D visualization of a limited number of image slices. In order to minimise the patient's X-ray exposure the number of slices is limited to 30-60 with a resolution of 256 x 256 per slice. A CT-image assigns a number to each point of the image. A range of numbers is characteristic for each component, whereby each number is a measur of density. That means at any given image point the corresponding number indicates e.g. muscles or bones. Interpolation algorithms try to find a hypothetical absorption coefficient in the interpolated image slices and decide for each component in the image (e.g. muscle, bone, etc), and where it should appear in the interpolated slice.

2.1.3 Representation of 3D medical objects and encoding algorithms

Modern medical imaging systems based on Computerized Tomography (CT), Nuclear Magnetic Resonance (NMR), Positron Emission Treatment (PET), or Ultra-sound scanners, generate large quantities of data sets, typically in the form of a series of 2D slices representing cross sections of human organs. If these 2D slices are stacked, a 3D representation is obtained that contains a wealth of information useful to the physician or medical researcher. There are

many potential applications of the 2D or 3D representation of human organs, including medical researcher , clinical diagnosis, surgical planning and radiation therapy [Cho93].

The most common form of these data sets is the 2D slice which represents a horizontal or longitidunal cut of a scanned human organ. Figure 2.3 represent a slice of human brain. It is recorded as a 128 by 128 matrix and is encoded in an BITMAP formatted file. This format will be explained later in this section.

Looking at the general field of 3D graphics, including CAD/CAM/CAE, Art and Flight Simulation, five major classes of data representation may be delineated. These are: Formatted 2D slices (PCX, BITMAP,etc...), Boundary based (B-rep), Constructive Solid Geometry (CSG), Parametric, and Spatial Enumerative (Voxel).

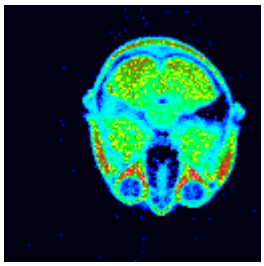


Figure 2.3: A 128 x 128 slice of a human brain stored as a bitmap encoded file.

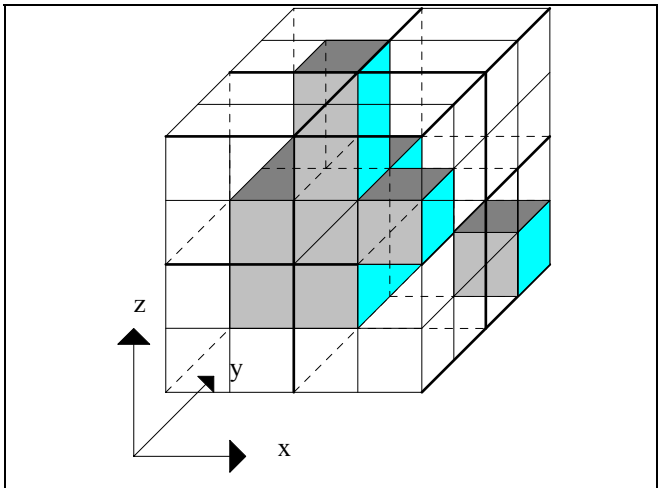
The formatted 2D slices describe a 2D surface in a raster fashion. The data encoding differ from one encoding technique to another (PCX, BITMAP, etc ...). But all the information concerning the slice such as it encoding technique and data pixels, slice dimensions and color map are stored in a structured header preceding the data section in the file record [datacomp95]

The boundary based methods (B-rep) describe only a set of surfaces, in terms of faces, edges and vertices. Faces may be polygons, or for example be constructed from B-splines (in 2D) or Bezier patches (in 3D) designed to maintain smoothness at boundaries in some specified way. The advantages of this approach lie in the existence of very well understood algorithms for

highly realistic images, and the availability of special purpose hardware for many of these [Fuch88]. The principal disadvantage is the lack of an internal representation; if a surface is cut or partially removed, there is no information about what is inside.

The Constructive Solid Geometry (CSG) representation is one composed of solid primitives, such as spheres, cylinders, cones and rectangular prisms. these are connected in a Boolean tree (usually a binary tree), which leads to general complex volumetric objects, which are amenable to attribute labelling as in 2D design graphics[Morris 88]. The disadvantages of this representation are due usually to the crudeness and limited number of original solid primitives.

The parametric technique represents the database as functions either every where in space or as surfaces or other manifolds in space [Levoy 88]. This technique is useful in graphic, Art and Engineering design and in mathematical or scientific applications such as the visualization of electron density maps in molecules, but limited in other areas unless used in combination with one of the other techniques.



Fgure 2.6: A simple object in a 4 by 4 by 4 cube.

The Spatial Enumeration technique (Voxel) describes space as a tessellation of unit cells, each containing attributes that may be very simple or quite complex. The simplest tessellation of space is provided by the cube [Meagher 82]. The unit cells are often called voxels, short for

volume elements in analogy to 2D pixels. In each voxel we store information in a binary form or multivalued, or greyscale.

In this section we will summarise some data structures used for representation of 3D objects in medical imaging. To fix the ideas a specific example of an object will be given. The object described is shown in figure 2.4. It is a binary object in a 4 x 4 x 4 array.

2.1.3.1 Boundary representations

1) 2D B-rep Approach : Only pairs of points and a depth for each slice need to be stored. Each set of points in at one depth is a contour.

2) 3D B-rep Approach: The 3D B-rep is produced from the 2D B-rep by tiling algorithm [Udupa87]. The output is a B-rep description of the surface. An algorithm called Marching Cubes uses an approach suggested by Cline [Cline88] where it directly construct tiles within voxels based on an estimate of whether the surface of the characteristic function intersects that voxel.

3) 2D Voxel Approach: In this representation, either pixel-locations are stored in order around the contour, or the outside and inside edges of the raster conversion of the enclosed region [Udupa82a].

4) 3D Voxel Approach: This representation is stored as lists, one for each orientation of the faces of a cubic voxel.

2.1.3.2 Volume representations

In this section, representations of a 3D digital array will be considered. A thorough review has been given by Srihari [Srihari81]. The representations considered are Digital arrays, Marginal Indexing, Run Length Encoded, Segment-Indexed Encoded (Segment End-point), and Slice-Based. The next section presents the octree representation.

1) Digital arrays

The basic structure of a 3D binary array is of the form: voxel-array[x][y][z]. If the object has been segmented from data, than voxels may values beeing either 1 or 0. In pratice the representation is often implemented by packing eight voxels to a byte[Freider85]

The binary digital array representation of figure 3.1 is given slice by slice:

Slice 0:
1 1 0 0
1 1 0 0
0 0 0 0
0 0 0 1

Slice 1:
1 1 0 0
1 1 1 0
0 0 0 0
0 0 0 0

Slice 2:
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 1

Slice 3:
0 0 0 0
0 1 0 0
0 0 0 0
0 0 0 0

2) Marginal Indexing

The 3D digital array can be implemented using marginal indexing, a pointer based representation [Srihari], rather than arrays of arrays. This representation is very convenient because, at the expense of a small increase in the storage requirement, agreater efficiency of access is achieved. The representation is dynamic and may be constructed by dynamic allocation:

```
1) Allocate an array of size Z, of slice-pointers
2) For each slice
        2.1) If the slice is empty
                Assign NULL to the slice-pointer
        2.2) Else
                Allocate an array of size Y, of row-pointers
3) For each row
        3.1) If the row is empty
                Assign NULL to the row-pointer
        3.2) Else
                Allocate an array of size X, of voxels
```

3) Run Length Encoded (Start End Point)

The run length encoding of binary digital scene is a list that stores the length of a run of 1-voxels and 0-voxels alternatively. For example the object of figure 2.4 would be

represented by: (0:8), (1:2),(0:2),(1:2),(0,1),(1:1),(0:8),(1:3),(0:1),(1:2),(0:11),(1:1),(0,6). The run-length encoding ignores the organization of the slices and rows. This encoding is very efficient in display techniques but it is not random access since a sequential access is required to read the value voxel(i,j,k).

4) Segment-Indexed Encoding

A method of using run-length encoding with a degree of random-access has been used by MIPG [Reynolds87]. Here a row of a slice is encoded as run length, but a marginal-index description is used to access the rows. The run-lengths are stored as start-end points referred to as line-segments, with a pointer to the next segment. This allows the storage to be dynamic. The technique leads to efficient storage and very fast display algorithms.

5) Slice-Based

It is possible to maintain only the slices, and to carry interpolation to access the voxels. This might be appropriate in a memory limited system [rhodes87].

2.1.3.2.1 The VDE file format

The Volumetric Data Encoding (VDE) is a file format for slice based representation of volumetric data. It is used in this thesis to load volumetric data from it. It has been developed by [Mougari& Laroui & Bouklachi 99]. The VDE file format holds the 3D data of a scanned organ such as head or kidney represented in a set of 2D slices.

The VDE is a fixed format, the file header contain information necessary to reconstruct the 3D object. The header structure of the VDE file header is given in program 2.1 bellow:

```
typedef struct file_header{
                    char file_type[6];  long file_size;
                    int user_header_size; int data_offset;
                    short int compression; short int gray_scale;
                    short int bit_per_voxel; int object_dim[2];
                    int voxel_dim[2];
                    char user_data[256]; char reserved[256];
                }VDEheader;
```

program 2.1: VDE file format (VDEheader).

Each field of the VDE header is explained the table bellow:

Table2.2 : VDE fields definition			
No	**field name**	**Description**	**Type**
1	file_type[6]	VDE.100	Char
2	file_size	File size	Long
3	user_header_size	User header size	Int
4	data_offset	offset of the first slice (raw data)	Int
5	compression	compressed or not	short int
6	gray_scale	color or grayscale	short int
7	bit_per_voxel	bits per voxel	short int
8	object_dim[2]	slice width, height, and #slices	int * 3
9	voxel_dim[2]	x,y,z of a voxel	int * 3
10	user_data[255]	user header related info	char * 256
11	reserved[255]	reserved for future use	char * 256

The VDE file format is described graphically in figure 2.5 . We notice that we are using only one header for N slices and slices have the same dimensions and are recorded with the same parameters.

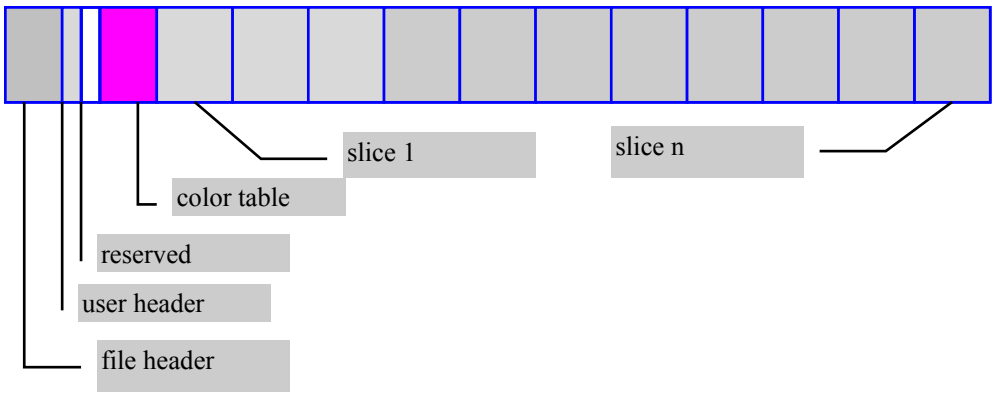


figure 2.5 : VDE file format

2.1.3.3 Octrees and Quadtrees

Thinking only of the display problem it is usually sufficient to provide an ordered list of those primitives (voxels or facets) that are to be displayed, with a mechanism

for running through the lists in an order appropriate for the viewing direction. However other applications might want to interrogate the data in a random way - for example to point to an object and read the data value there, or to give quantitative distance values, or in surgical simulation to detect interference as parts of structures are moved with respect to each other. In such circumtances it is highly advantageous to have random access into the data. A three-dimensional array of voxels or an octree structure allows random access, whereas run length encoded structutres do not.

A k-tree, where $k=2^n$, is a representation of n-dimensional space. Thus a quadtree is a symmetric, recursive, hierarchical description of an image. The octree is its 3D extension, and it is a hierarchical representation with nodes that have a status (homogeneous or inhomogeneous). If we consider that the term octree will refer to a symmetric recursive description[Srihari81]. Figure 2.7 shows this representation applied to the object in figure 2.4.

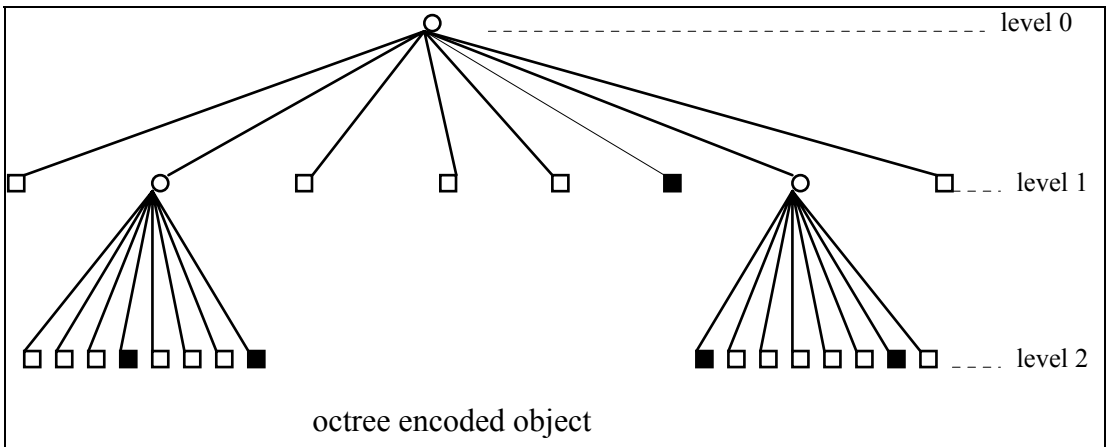


Figure 2.7: The symmetrical representation of the object in figure 2.6

The information stored at a node characterises the type of tree. Therefore, octree structures that store information at inhomogeneous nodes are deemed pyramid trees. The information at a node may be binary, multivalue or greyscale. The most appropriate encoding method is a subject of much discussion. A good summary is provided in [SAMET88a].

Many authors have pointed out that octree/quadtree algorithms often have a time complexity linear in the number of nodes in the tree [SAMET88a]. The basic operations on octrees are

boolean and bilinear transformations. Other simple operations are reflection, rotation by 90 degrees, and scaling by a power of two. In some algorithms an octree or quadtree node will have to be condensed (merged) or spawned (split). A linear octree can be searched by binary search. Treecode will require a sequential search. An explicit tree can be randomly accessed. A related process is neighbour finding, which is a requisite in many algorithms.

2.1.3.3.1 Octree and Quadtree data structures

The human organs do not have regular shapes, therefore the octree representation is suitable to encode them in the image space rather than in the object space. For instance one voxel could be a part of a skin tissue while a close voxel might be a part of a bone or an empty region. The octree encoding technique is a suitable encoding technique for medical imaging because we can partition the human organ into regular sub-cubes and attribute region states to each and save time in the display by considering only the homogeneous regions as it is shown in figure 2.4.

Pixels and voxels data types (Point2D & Point3D)

The Point3D is a data type that represents a three dimensional point in the space. It has three fields, the x,y, and z coordinates. Using the C++ language structure, we declare this type as follows:

```
typedef struct point_3D {      int x;
                               int y;
                               int z;
                           }Point3D
```

program 2.2: Point3D

The Point2D is a data type that represents a two dimensional point in the plane. It has two fields, the x and y co-ordinates. We declare this type as follows:

```
typedef struct point_2D {      int x;
                               int y;
                           }Point2D;
```

program 2.3: Point2D

The octree which represents the volume has nodes called octants. These octants represents sub-volume delimited by an Upper and Lower voxels as shown in figure 2.8. The octree is the data type that represents the node of the octree tree. The node contains five members that are Lower, Upper, Hmg, Color, and *Octant[8].

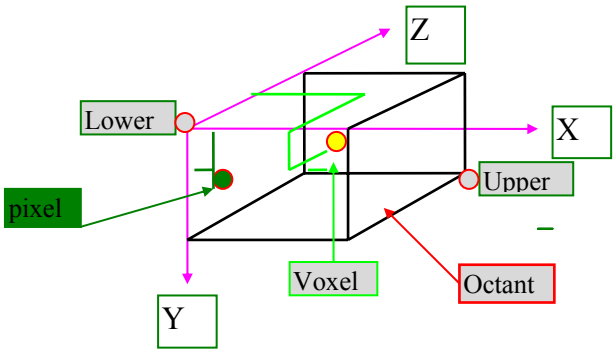


Figure 2.8: Octant Lower and Upper limits.

Lower and Upper are of type 3D point, Lower bounds the lower limits of specific volume while Upper bounds the upper limits. Hmg is a flag to indicate the state of the octant. Color will hold the color of a homogeneous octant. Finally *Octant[8] is an array of eight pointers to link the kids to their parent node. This type is declared as follows:

```
typedef struct octree{ Point3D Lower;
                       Point3D Upper;
                       int Hmg;
                       int Color;
                       struct octree *Octant[8];
                       }Octree;
```

program 2.4: Octree data type

The Quadtree is the data type that represents the node of the quadtree tree. It contains five fields that are lower, Upper, Hmg, Color, and *quadrant[4]. Lower and Upper are of type Point2D, Lower bounds the lower limits of a quadrant while Upper bounds the upper limits. Hmg is a flag to indicate the state of the quadrant. Color will hold the color of the quadrant in the case of a homogeneous state. Finally *quadrant[4] is an array of four pointers to link the kids to the parent node. This type can be declared as follows:

```
typedef struct quadtree{
                    Point2D Lower;
                    Point2D Upper;
                    int Hmg;
                    int Color;
                    struc quadtree *quadtrant[4];
                }Quadtree;
```

program 2.5: Quadtree data type

Octreeptr and Qudtreeptr are defined types, and they are pointers to Octree and Qudtree respectively. They are declared as follows:

```
typedef Octree *Octreeptr; and
typedef Qudtree *Quadtreeptr;
```

2.1.4 SEGMENTATION ALGORITHMS

2.1.4.1 Image segmentation

An image as it is stored by a computer is just a multidimensional array of pixel values. Although we as humans may look at the displayed image and recognize it as meaningful, the computer must algorithmically analyse the array of pixel values before it can reach any conclusions about the content of the image.

A computer must deal with objects in an image, as objects and not just as unrelated pixels, for many reasons:

1) computer vision (e.g., robotics);

2) computer analysis of quantitative properties of objects;

3) computer manipulation of objects for image display via man-machine interactions;

4) object-based nonstationary image restoration.

Before any of the above mentioned object-related actions can be taken, one or two conditions must be met:

1) The object must be recognized as an entity distinct from other objects in the image (i.e., pixels belonging to the object must be understood to be related in some way).

2) The entity must be labelled. It must be understood that it is, in fact, the specific object that is being searched for.

Most image processing techniques perform the step one first and independently of step two. Step 1 is commonly called the image segmentation.

There exists a large number of image segmentation techniques. Most techniques, however, fall into one of the three broad categories: region growing, boundary detection, or multiresolution. Multiresolution techniques (e.g., the Pyramid, the DOLP transform, complete trees, edge focusing [Jain89]), attempt to gain a global view of an image by examining it at many different resolution levels. The lower resolution provides a global view of the image, and the higher resolution provides the detail. The stack approach proposed by Koenderink is a multiresolution image description and segmentation technique. The stack calculates image segments and an image description tree by associating every pixel in an image with a local intensity extremum. The blurring kernel reflects the shapes of the regions of interest and yield segmentation results by performing ,anisotropic stationary blurring and nonstationary blurring.

Theoretical calculations, simulations, and application to actual images all clearly show that isointensity paths are able to move more quickly in the direction in which blurring is faster. In addition, some paths which before headed away from an extremum may now head towards it, thus the extremal region associated with an extremum is clearly different in the anisotropic case from the isotropic blurring case.

Spatially variant blurring based upon local image content should improve the correspondence between the extremal regions and the semantically meaningful regions in the image. This statement is motivated by the human visual model of Cohen and Grossberg, which indicates that object perception is performed via an intensity diffusion process (i.e.

blurring) which is nonstationary). There is no simple coordinate transformation between embedding produced by nonstationary blurring schemes and stationary blurring schemes.

2.1.4.2 Edge detection algorithms:

Current literature abounds in a variety of edge detection algorithms. The edge detection plays an important role in number of image processing applications such as scene analysis and object recognition. The edge in an image provide useful structural information about object boundaries, as the edges are caused by changes in some physical properties of surfaces being photographed, such as illumination, geometry, and reflectance. An edge is, therefore, defined as a local change or discontinuity in image luminance.

There are two basic approaches to edge detection: the enhancement/thresholding method and the edge fitting method. In the former method, the discontinuities in the image gray tone are enhanced by neighborhood operators. The simplest type of differential operator, the Robert's operator, was introduced more than three decades ago. This operator consist of 2 x 2 convolution kernels. Prewitt and Sobel operators are improvements on Robert's operator, where the masks have sizes 3 x 3 pixels. The Sobel operator involves the computation of local intensity gradients and the responses due to nonideal step edges are not good. A modification of that is the detection of second-order zero-crossings, and the corresponding edge operator is called the Laplacian. In order ot reduce the effect of high frequency noise, a bandwidth restriction can be imposed by convolving the original data by a two-dimensional Gaussian window before applying the Laplacian operator [Marr80].

Algorithms belonging to the second category, the edge fitting method, minimize the distance between the original noisy data and a predefined edge model in a finite-dimensional space. Thus the results of these algorithms are optimal for the chosen edge model and are less sensitive to noise. However, they involve more computations. One such algorithm that recognizes edges and lines (edges of finite thickness) was introduced by Hueckel.

As an application of edge detection we can consider the detection of blood vessels in retinal images using two-dimensional matched filters. Information about blood vessels can be used in grading disease severity or as part of the process of automated diagnosis of diseases with ocular manifestations. Blood vessels can act as landmarks for localizing the optic nerve, the fovea (central vision area), and lesions. As a result of systemic or local ocular disease, the blood vessels can have measurable abnormalities in diameter, color, and tortuosity. For example, central retinal artery occlusion usually causes generalized constriction of retinal arteries, hypertension may result in focal constriction of retinal arteries, hypertension may result in focal constriction of retinal arteries, central retinal vein occlusion typically produces dilated tortuous veins, arteriosclerosis can cause the arteries to acquire a copper or silver color, and diabetes can generate new blood vessels (neovascularization). Thus, a reliable method of vessel detection is needed, which preserves various vessel measurements.

An edge detection algorithm, similar to the morphological edge detector, has been used for automatic recognition of arterio-venous intersections[edge]. A line finding algorithm along with a probabilistic relaxation scheme has been used for the extraction and subsequent description of blood vessel patterns in retinal images [Chaudhuri89].

2.1.4.3 The elastic matching algorithm (mssm)

It is not possible to enumerate all the algorithms involved in all medical imaging applications. However we have selected one of the very important algorithms that enters in the medical diagnosis is the mssm algorithm to discover abnormalities in human organs by comparing them to an anatomical model. However the principles used in the algorithm are multiresolution filtering and elastic matching which are used in many other applications such signal processing and vision in general. This algorithm will be studied in depth in this thesis and many parallel processing techniques will be applied to its parallelization.

The Multiple Scale Signal Algorithm

The Multiple Scale Signal Matching algorithm (MSSM)[Mowfoth89] performs the matching in a pipeline of **num_stages** stages, each stage has two processing steps as shown program 2.1. The first step is filtering, which involves filtering the **organ_slice** and the **model_slice** by a set of separated difference of Gaussian filters **dog()**. The Gaussian filters (bandpass) are created with a set of variance values: ***σ(1), σ(2), ..., σ(i) ,..***. Where the variance ***σ(i)*** is the parameter which determines the lowest frequency of the band-pass filter (Its value is determined by gradient, following Marr's convention concerning the largest disparity [Marr82] the process is intended to match). The second step is matching **match()** or correlation. This process searches for each pixel in the image for the closest matching pixel in the model. This process involves a window of nearest neighbourhood pixels of dimension (**window_size**) . The result of this match is a new discrepancy map which is subsequently used in the next stage in the pipeline with a finer Difference of Gauss filter. The discrepancy map **discrepancy_map** that results from the final matching stage is then used to estimate the original image-model discrepancy.

```
MSSM (organ_slice,model_slice,discrepancy_map,num_stages,window_size)
{
  for( i = num_stages; i > 0 ; i--)
    {
     /*Step1: filtering */

     filtered_organ_slice = dog (organ_slice, σ(i), σ(i+1));
     filtered_model_slice = dog (model_slice, σ(i), σ(i+1));

     /*Step2: matching */

     for(j = num_of_pixels_in_image; j > 0 ; j--)
        new_discrepancy_map =  match (filtered_organ_slice[j],
                                      filtered_model_slice,
                                      previous_discrepancy_map,
                                      window_size);

     previous_discrepancy_map ← new_discrepancy_map;
    }
}
```

Program 2.1 :MSSM algorithm.

The discrepancy maps that resul can be considered as an input to a segmentation stage to extract tissues of interest from the data (i.e. blood vessels or bones) as shown in figure 2.9(a).

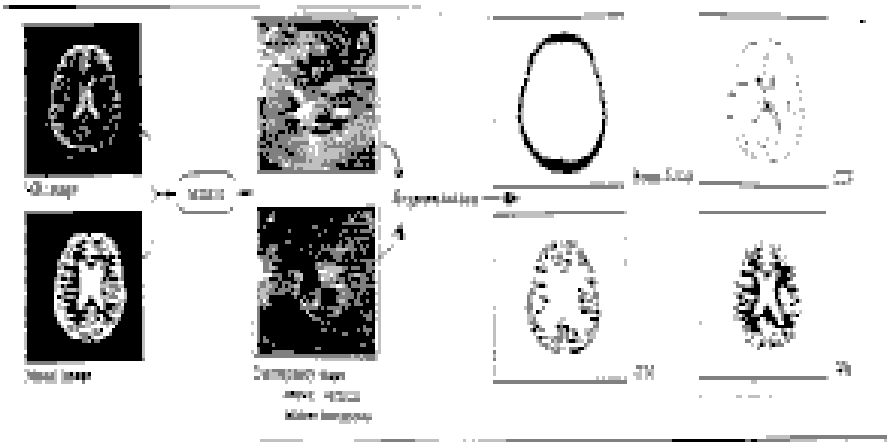


Figure 2.9(a): The mssm algorithm used to extract tissue of interests.

2.1.5 Visualization algorithms:

Surface-based and binary voxel techniques for displaying sampled scalar fields of three spatial dimensions have been in use for about ten years and have been applied to a variety of scientific, medical, and engineering problems.

During the last decade, the problem of low image quality has been largely solved with the development of volume rendering. Volume rendering is a technique for visualizing sampled scalar fields of three spatial dimensions without fitting geometric primitives to the data. A color and a partial transparency are computed for each data sample, and images are formed by blending together contributions made by samples projecting to the same pixel on the picture plane.

2.1.5.1 Volume rendering

Despite its advantages, volume rendering has proven to suffer from a number of problems. High on this list is the technique's computational expense. Since all voxels participate in the generation of each image, rendering time grows linearly with the size of the dataset. Published techniques take minutes or even hours to generate a single view of a large

dataset using currently available workstation technology. Other algorithms exploit the spatial coherence in the 3D data and in its 2D projections to reduce the computational expense of the volume rendering algorithm. Three techniques for reducing rendering cost are presented in [Levoy89] based on hierarchical spatial enumeration, adaptive termination of ray tracing, and adaptive image sampling with case studies from two applications: medical imaging and molecular graphics.

In many applications, the data to be visualized has no visible manifestation in nature. Internal anatomic surfaces are visible during surgery, but seldom in their entirety and seldom under the lighting and viewing conditions simulated in volume rendering algorithms. If we treat volume rendering as abstract visualization, adherence to a consistent physical model is not necessary. On the other hand, the human perceptual system expects sensory input to arise from physically plausible phenomena and forms interpretations on that basis.

The near-term prospects for real-time volume rendering are encouraging. Kaufman has written an excellent survey of architectures designed for rendering voxel data [Kaufman86a], including his own CUBE (Cubic frame Buffer) system currently under development [Kaufman88a]. Most of the machines he surveys start by thresholding the incoming data. At present, the fastest system based on volumetric compositing is probably the Pixar Image Computer. Using a four channel SIMD processor[Levinthal84] and Pixar's ChapVolumes software package, the Image Computer can generate high-quality images in terns of seconds or minutes, depending on the size of the dataset. Algorithms for ray tracing geometrically defined scenes have been developed for a number of parallel machine architectures including the Connection Machine[Delaney88] and the MPP [Dorband88], but none of these implementations support the display of volume data. A parallelizable object-order volume rendering algorithm is presented in [Westover89], but no implementation achieving real-time update rates has yet been attempted.

To display medical objects such as the organ shown in figure 2.9 , this later has to be represented in one of the representations presented earlier. All of the representations have been used to some extent in medical imaging, although the most common are the Boundary representation and the Spatially Enumerative methods, because of their natural correspondence to the data.

2D display of 3D objects are referred to as 3D displays. This type of display may be divided into surface rendering and volume rendering algorithms. In the surface based methods, the object is represented by its surface, which is defined as the set of all the voxel faces which separate voxels in the object from the bacground. 3D surface tracking algorithms have been developped to find this set of faces directly from a segmented digital data volume (DV). In these surface based methods, visualization is usually done by a Z-buffer algorithm.

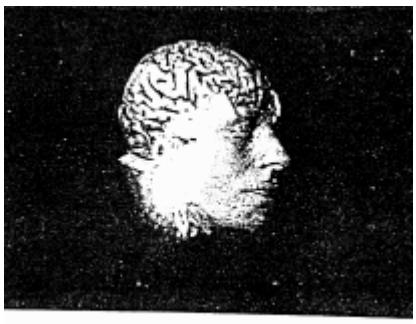


figure 2.9: volume rendering of a human head.

In volume based methods, the object surface is not searched for. The object is described by a set of volume elements (rectangular parallelepipeds) in a spatial occupancy enumeration scheme. Two data structures can be used for this description, a 3D array of voxels or an octree. Special surface visualization methods have been designed for both representations. The visualization algorithms Back-To-From (BTF), Front-To-Back(FTB) and Ray Tracing have been implemented on special purpose hardware and processors and achieved real-time display.

In recent years, ray tracing has become the algorithm of choice for generating high fidelity images. Its simplicity and elegance allows one to easily model reflection, refraction and shadows. Unfortunately, it has a major drawback: computational expense. The prime reason for this is that the heart of ray tracing, intersecting an object with a ray, is expensive and can easily take up to 90% of the rendering time. Unless some sort of intersection culling is performed, each ray must intersect all the objects in the scene, a very expensive proposition. There are two general strategies for intersection culling: hierarchical bounding volumes and space partitoning.

One of the principal drawbacks of the volume rendering algorithm is its cost. Since all voxels participate in the generation of the image, rendering time grows linearly with the size of the dataset. There exist two techniques for reducing the expense of tracing a ray through volume data.

The first optimization is based on the observation that many datasets contain coherent regions of emplty voxels. in the context of volume rendering, a voxel is defined as empty if its opacity is zero. Techniques for encoding coherence in volume data include octree hierarchical spatial enumerations [Meagher82], polygonal representations of bounding surfaces [Fuchs77, Pizer86], and octree representations of bounding surfaces [Gargantini86].

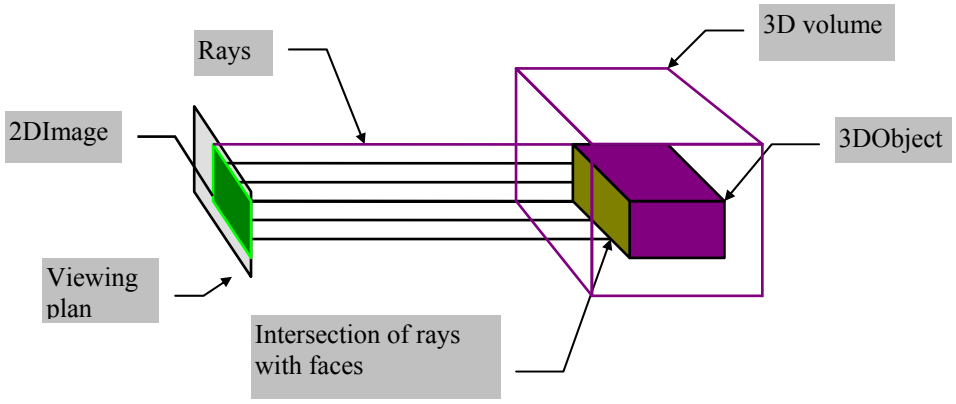


Figure 2.10: Raycasting process

The second optimization is based on the observation that once a ray struck an opaque object or has progressed a sufficient distance through a semi-transparent object, opacity accumulates to a level where the color of the ray stabilizes and ray tracing can be terminated.

2.1.5.2 Octree visualization method

When the octree representation is used, the hidden surfaces are eliminated by chosing the frontal nodes 0,1,2, and 3 to be displayed. By this, we are viewing surfaces in front-to-back order. In figure 2.10 the front face is formed with the faces 0,1,2, and 3, while the back face is formed with faces 4,5,6, and 7. Considering a node, the octants 0,1,2, and 3 are visited before the octants 4,5,6, and 7. This continues for each node subdivision. When having a homogeneous state of a node, the recorded color should be buffered. Hence, it is obvious that a quadtree is needed to map this octree into a 2D encoding scheme. After mapping the octree to a qudtree the 2D image is then easily constructed.

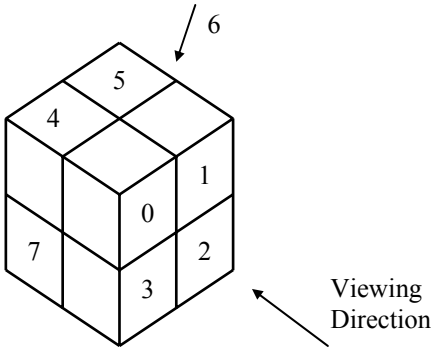


Figure 2.11: Objects in octants 0, 1, 2, and 3 obscure objects in the back octants (4,5,6, and 7) when the viewing direction is as shown.

The octree visualization algorithm.

```
/*****************************************************************/
/*          O C T R E E  V I S U A L I Z A T I O N  A L G O R I T H M      */
/*****************************************************************/
                        /* Data type declarations and Variables */
const
  EMPTY = -1;
type
```

```
  octNodePtr = ^octNode;
  octEntry = record
    case homogeneous : boolean of
      true : (color : integer);
      false : (child : octNodePtr)
    end;
  octNode = array [0..7] of octEntry;
  quadNodePtr = ^quadNode;
  quadEntry = record
    case homogeneous : boolean of
      true : (color : integer);
      false : (child : quadNodePtr)
    end;
  quadNode = array [0..3] of qudEntry;
var
  octree : octNode;
  quadtree : quadNode;
  newquadtree :qudNodePtr;
  backcolor : integer;
```

assumes formal view of octree (with octants 0, 1, 2, and 3 in front) and converts it to quadtree. Accepts octree as input, where each element of the octree is either a color value, the EMPTY flag, or a pointer to a child octant node.

```
procedure octToQuad (octree: octNode; var quadtree : quadNode);
var
  k : integer;
begin
  for k := 0 to 3 do
    begin
      quadtree[k].homogeneous := true;
        if octree[k].homogeneous then
          if (octree[k].color > EMPTY) then
            quadtree[k].color := octree[k].color
          else
            if octree[k+4].homogeneous then
              if (octree[k+4].color > EMPTY) then
                quadtree[k].color := octree[k+4].color
              else
                 quadtree[k].color := backcolor
            else
              begin
                quadtree[k].homogeneous := false;
                 new (newquadtree);
                 quadtree[k].child := newquadtree;
                octToQuad (octree[k+4).child^,newquadtree^)
               end
          else
            begin
              quadtree[k].homogeneous := false;
              new(newqudtree);
              quadtree[k].child := newquadtree;
```

```
            octToQuad (octree[k+4].child^, newquadtree^);
            octToQuad (octree[k+4].child^, newquadtree^)
          end
      end
end;
```

Program: 2.2: Octree displaying algorithm.

2.2 Medical imaging systems, software tools and their computational requirements

2.2.1 Medical imaging computational requirements

Studying the requirements of medical imaging systems and of their medical applications reviel two inportant aspects. The first aspect is related to the computational complexity and heavy load of the algorithms involved in various imaging tasks. The second aspect is related to the real time requirement of most medical applications (i.e. emergencies). Processing medical information is very computationally demanding and require powerful computer systems. For this reason various strategies have been followed to conceive medical imaging systems. We have tried to list some of their aspects in the following points:

- Compact and efficient software packages designed to run on the hardware of the data acquisition device such as the 3D98 at University of Pensilvania at Philadelphia.
- Portable software packages designed to run on the stand-alone dedicated computing platforms such as the ANALYSE marketed by CTI (Tennesse) at MAYO clinic, Biodynamics Research Unit, Minneapolis, Minnesota [Robb89].
- Special purpose hardware such as the INSIGHT which is based on the VOXEL PROCESSOR [Meagher85] .
- Concurrent software such as the BODYSCAN which runs on a concurrent supercomputer [Mills90].

2.2.2 Approaches to the use of 3D graphics in medical imaging

Table 2.3 summarises the research group, The university or the company of the original research group, the name of the researchers, what have been pionnering in, the development they have produced, and finaly adequate references.

Table 2.3: Approaches to the use of 3D graphics in medical imaging.

Group	Origin	Researchers	Pionnering work in:	Development	References
MIPG	University of Pensilvania at Philadelphia	Prof. G.T. Herman	Voxel databases	Com. Package 3D83-98, FORTRAN, S100 Proc., GE9800 CT scanner	ARTZY81, HERMAN83 UDUPA86
Vannier & Marsh	Mallinckdrot Institute, ST. Louis, Missouri.	Vannier & Marsh	Interactive surgical planning	The commercial package Siemens 3DCT, Fortran, PDP11-34 processor of the Siemens Somatron CT scanner.	VANNIER83 TOTTY84
CEMAX	Contour Medical Systems at Santa Clara California	Commercial company	Interactive surgical planning	CEMAX1000, 68000 based comp. Platform	DEV84 FELLING HAM86
UCL3D	University College London	A DHSS funded project	Appl. of computer graphics to surgical planning	A software package (UCL3D) implemented on a 32 bit super-mini (ND540)	ARRIDGE89
MEAGHER	Rennsalear Polytechnic Albany, N.Y.	Meagher currently president of Octree corpo.	He developed the original volume based approach to 3D medical graphics.	Developed the special purpose hardware system INSIGHT	MEAGHER 82, 83, 84c, 84d, 85.
Goldwasser & Reynolds	Dynamic Digital 3D Inc. Phelidelphia MIPG coll.	Goldwasser and Reynolds	Formed a company: Dynamic Digital Display (3D) (phdelphia)	VOXEL PROCESSOR special purpose hardware.	GOLD WASSER 85, 86, 87.
UNC	The university of North Carolina at Chapel Hill	Henry Fuchs	World expert VLSI tech responsible for VLSI polygon renderer	The VLSI polygon renderer PIXEL-PLANES	FUCHS 88
Hamburg	University of Hmburg	K.H. Hohne	Greyscale ray tracing techniques 3D grad op. for shading	The commercial system marketed by KONTRON (Munich)	HOHNE
PIXAR	Commercial company based at san Rafael California	Lucas Film graphic art company (Star Wars)	Volume rendering	special purpose hardware purpose hardware device using an approach called volume rendering	FISHMAN
MAYO	The Mayo cinic Minneapolis Minnesota	Robb	Fast volume based methods to inspect the large quant. of data (DSR)	Special purpose hardware, Software package ANALYSE marketed by CTI (Tennesse)	HEFFERNAN 85b, ROBB 88.

2.2.3 Medical graphics systems

Table 2.4 lists some products of medical graphics systems.

Table 2.4 states the type of the medical graphics system in a very short description. However we are presenting in the next section the ANALYSE software package.

Table 2.4: Medical Graphics systems	
Product	**Type**
3D89-3D98	Commercial package
3DCT	Commercial package
CEMAX 1000	68000 based computing platform
UCL3D	Software package implemented on a super- minicomputer.
INSIGHT	Special purpose hardware system
VOXEL PROCESSOR	Special purpose hardware system
PIXEL- PLANES	VLSI polygone hardware
KONTRON	Commercial system
PIXAR	Special purpose hardware
ANALYSE	Portable software package
BODYSCAN	Software from EPCC, runs on concurrent supercomputer.

2.2.3.1 The ANALYSE software package

A comprehensive software package, called ANALYSE, has been developed [Robb89] which permits detailed investigation and evaluation of 3-D biomedical images. ANALYSE can be used with 3D imaging modalities based on X-ray computed tomography, radionuclide emission tomography, ultrasound tomography, and magnetic resonance imaging. The package is unique in its synergistic integration of fully interactive modules for direct display, manipulation, and measurement of multidimensional image data.

One of the most versatile and powerful capabilities in ANALYSE is image volume rendering for 3D display. An important advantage of this technique is to display 3D images directly from the original data set and to provide ” on-the-fly” combinations of selected image transformations, such as surface segmentation, cutting planes, transparency, and/or volume set operations (union, intersection, difference, ect...). This module is optimized to be fast (interactive), without compromising image quality. The software is written entirely in C and runs on standard UNIX workstations. The system provides an effective shell for prototyping

custom software and turnkey applications. The software architecture permits systematic enhancements and upgrades which has fostered development of a readily expandable package.

The Biodynamics Research Unit (BRU) at Mayo Clinic has been involved in the analysis of 3D and 4D images since early 1970's. The successful development of the dynamic spatial reconstructor has provided a ready source of 3D and 4D imagery from a variety of biomedical investigative and clinical studies. During this time, several unique and useful analysis techniques have been developed in the BRU. These include projection displays, stereo displays, interactive oblique sectioning algorithms, shaded surface display algorithms, and space-filling virtual 3D display using a vibrating mirror system. Many of these capabilities have been developed on different computer systems as the need for them arose. Recently, these established tools and several new important capabilities have been generalized and made applicable to a wide variety of multidimensional biomedical images (e.g., CT, MRI, PET, ultrasound, ect.) and have been integrated into a single comprehensive software package which runs on a dedicated, powerful image analysis workstation. To support this application and other projects an integrated network of powerful workstations has been developed. Figure 1 is a diagram of this architecture, which is called BRU-NET. Three major divisions (LAN's, or local area networks) of a variety of workstations are, respectively, focused on ultrasound, DSR, and advanced algorithm research. The system include powerful workstations from Sun Microsystems and Silicon Graphics, as well as advanced image processing systems from Pixar and a robust number cruncher from Masscomp.

All the components in the computer network run under a standard operating system (UNIX), are connected by standard Ethernet cable, and have compatible network file communication protocols (TCP/IP) running under the standard network file server (NFS) system. The overall system can readily grow or shrink simply by adding or deleting components to and from the network as appropriate. Workstations are configured so as to

function independently of the network if desired, but can benefit from availability of other resources on the network if connected to it. The file servers can be readily expanded to support an increased number of users and/or programmers.

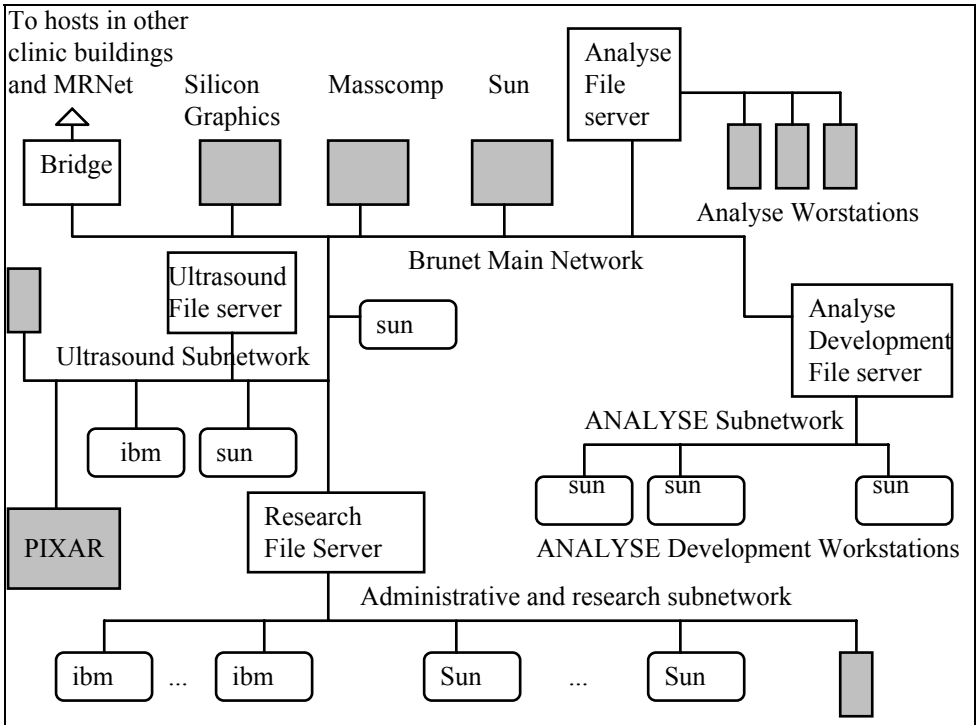


figure 1.x: Network of multidimensional image analysis systems in Mayo Biotechnology Computer Resource.

2.2.3.2 The VDE software

In this section we will present the design and implementation of the VDE imaging software [Mougari, Laroui, and Bouklachi 99]. The VDE software has been designed to encod ordinary slices into a VDE file format (refer to section 2.1.3.2.1).

The VDE software runs under windows95 through a powerful user interfaceand provide the user with a set utilities to encode and decode from /to VDE file formats to octrees and perform some image processing tasks (filtering and segmentation). It is also provided with an error handler modules.

The main modules of the VDE software are: encoding, decoding, image processing, octree module, and error handler module.

Encoding module: This module include the functions to save a VDE file: The first function is decimation. Provided with a cube , this function reduce its size by filtering and subsampling the cube data. The seconde function is save_vde_file, which saves data to VDE file format. The last function is RLE_encode that encodes each slice in the cube using the RLE encoding algorithm.

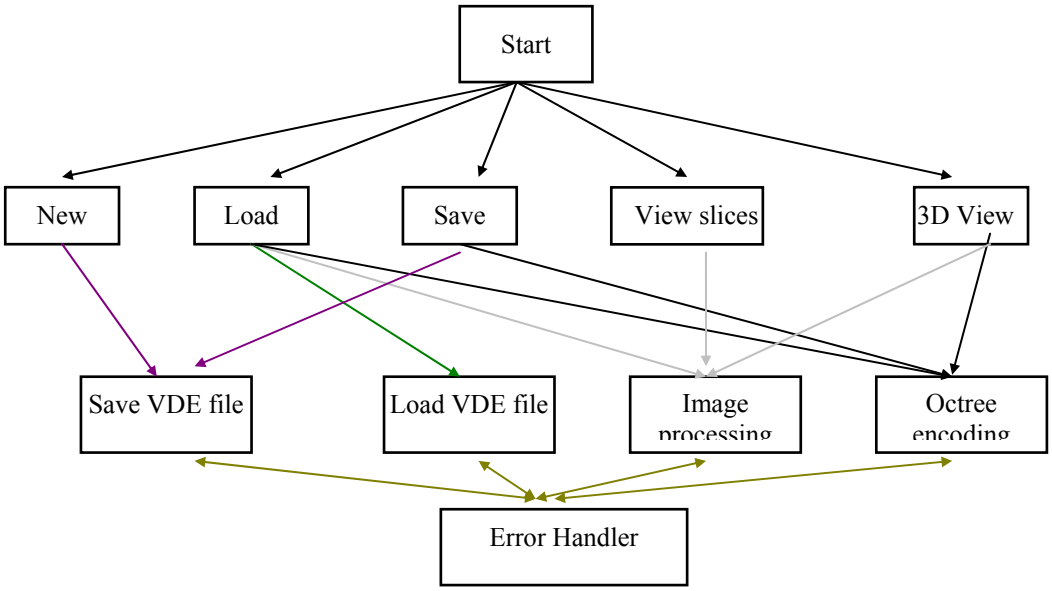


Figure 2.9: VDE software

Decoding module: This module comprises the load_vde_file function that reads data from a VDE file. Then using image interpolation and filtering increases the number of object slices resulting in a higher resolution.

Image processing module: This module comprises the following functions: convolution, interpolation, convert to 24 bit per pixel, convert to 8 bit per pixel, convert to a gray scale image.

Octree module: This module comprise the following functions: Build octree, test octants, octree to quadtree, traverse octree, traverse quadtree.

Error handler module: This module handles errors and generate error codes and resumes from failure.

Chapter 3a

3a Parallel computers

3.1 Parallel computers

3.1.1 Introduction to parallel computers and distributed memory multicomputers

3.1.1.1 pipeline computers

3.1.1.2 Vector computers

3.1.1.3 Multiprocessors

3.1.1.4 Distributed memory multicomputers

3.1.2 Transputers and related systems

3.1.2.1 Transputer architecture

3.1.2.2 Support for concurrency

3.1.2.3 The INMOS link

3.1.2.4 Performance

3.1.2.5 Building systems with transputers

3 Parallel computers, performance measures, models of parallel computations

3.1 Parallel computers

3.1.1 Introduction to Parallel computers and Distributed Memory Multicomputers

Parallel computers are those systems that emphasize parallel processing. The basic architectural features of parallel computers are introduced below. We divide parallel computers into three architectural configurations:

- Pipeline computers
- Array processors
- Multiprocessor systems

A pipeline computer performs overlapped computations to exploit temporal parallelism. An array processor uses multiple synchronized arithmetic logic units to achieve spatial parallelism. A multiprocessor system achieves asynchronous parallelism through a set of interactive processors with shared resources (memories, database, etc.). These three parallel approaches to computer system design are not mutually exclusive. In fact, most existing computers are now pipelined, and some of them assume also an *array* or a *multiprocessor* structure. The fundamental difference between and array processor and a multiprocessor system is that the processing elements in an array processor operate synchronously but processors in a multiprocessor system may operate asynchronously.

3.1.1.1 pipeline computers

Pipelining is an approach to relieve bottlenecks in high speed computers. The idea is to use parallelism at the point of the bottleneck to improve performance globally. If the design techniques are successful, then the extra hardware devoted to performance enhancement is present in only a small portion of a computer system, yet its effect is to improve performance as if the full computer system were replicated. The speed increase is obtained by clever

architectures which create efficient computer systems in which enhancements of a relatively small cost have a global impact on performance.

In the 1960s, when hardware costs were relatively high, pipelined computers were the supercomputers. IBM's STRETCH and CDC's 6600 were two such designs from the early 1960s that made extensive use of pipelining, and these designs strongly influenced the structure of supercomputers that followed. By the 1980s, hardware costs had diminished to the extent that pipeline techniques could be implemented across the entire range of performance, and indeed, even the Intel 8086 microprocessor that costs just a few dollars uses pipeline accesses to memory while performing on-chip computation.

3.1.1.2 Vector computers

Most vector computers have a pipelined structure. When one pipeline is not sufficient to achieve desired performance, designers have occasionally provided multiple pipelines. Such processors not only support a streaming mode of data flow through a single pipeline, they also support fully parallel operation by allowing multiple pipelines to execute concurrently on independent streams of data.

By the mid-1980s, more than twenty manufacturers offered vector processors based on pipeline arithmetic units. they ranged from relatively inexpensive auxiliary processors attached to microcomputers to high-speed supercomputers with computation rates from 100 Mflops to rates in excess of 1000 Mflops. (One Mflops is 10^6 floating-point operations per second.)

The price-performance ratio of these vector processors is rather remarkable because they yield one to two orders of magnitude increased throughput for vector computations when compared to serial coprocessors of equal cost. But this throughput increase is limited to the problems that fit the architecture-that is, to problems that can be structured as a sequence of vector operations whose characteristics make efficient use of the facilities available.

Many of the supercomputers are high-performance serial processors for general-purpose problems, but the throughput of these supercomputers on non-vector problems is only a few times greater than the throughput of more conventional high-speed serial processors. In fact, although throughput might be high because of fast device technology, if a vector-structured supercomputer is used exclusively on nonvector problems, the computational cost may be excessive because this cost includes the cost of the vector facilities, which presumably are left idle by scalar computations.

Examples of some vector computers are listed below

- Variable delays in the I/O streams and pipelined arithmetic unit CDC STAR , a combination between pipeline and fully parallel implementation)
- Hierarchical memory structure (intermediate memories) such as the Cray I.
- The data flow and processor/memory structure such as the Burroughs Scientific Processor.
- Attached Vector-Processors such as the FPS-164 attached processor
- IBM developed another combination between pipelined and parallel design with a much stronger parallel component than the FPS-164 has. The GF-11 is very much like a richly connected ILLIAC IV.

3.1.1.3 Multiprocessors

The motivation for moving toward multiple processors is strictly a matter of performance because device technology places an upper bound on the speed of any single processor. To exceed that bound requires multiple processors.

Our earlier discussions of high-performance machines study two important classes of parallelism. Pipeline machines produce high performance by placing several stages of a pipeline in operation simultaneously. Machines for continuum calculations have multiple processors, each executing the same program. In both cases, a single program is used to operate on vectors or arrays of data. Flynn [1966] termed this type of parallelism single-

instruction stream, multiple-data stream (SIMD) parallelism. Recall ,for example, an extreme implementation of this idea in the form of the GF-11, in which each of 576 processors executes identical instructions broadcast to them by a single control unit.

Another SIMD machine with massive parallelism is the Connection Machine [Hillis 1986] with 64K 1-bit processors. The architect is truly fortunate when an application can be executed on machines that are built around the lock-step parallelism required for SIMD machines because the architecture efficiently executes programs well suited to SIMD execution.

High performance on such machines requires rewriting conventional broadcast to all processors. Although programming for these machines can be difficult in principle, in the ideal case, a serial algorithm can be converted to an SIMD algorithm by replacing each inner loop with a single broadcast instruction that implements the complete loop. The fact that an important, but limited, class of problems fits this model extremely well has provided the impetus for the design and construction of these machines.

Clearly, some large problems do not lend themselves to efficient execution in an SIMD architecture. The operations required for such problems cannot easily be organized into repetitive operations on uniformly structured data. They tend to be unstructured and unpredictable. Addressing patterns tend to be data dependent, so the architecture cannot easily preload data by anticipating future accesses.

The architect who must attain high performance for such problems inevitably looks for a solution in a multiprocessor structure. Such an architecture is composed of several independent computers, each capable of executing its own program. Flynn [1966] calls this type of architecture multiple-instruction- stream, multiple-data stream (MIMD) architecture. The processors of a multiprocessor are interconnected in some fashion to permit programs to exchange data and synchronize activities.

A model of such an architecture is shown in figure 3.1. In figure 3.1(a) the memory is shared but in figure 3.1(b) the memory is attached to individual processors. In both cases, because the system contains multiple processors, each capable of executing an independent program, the system fits Flynn's MIMD model.

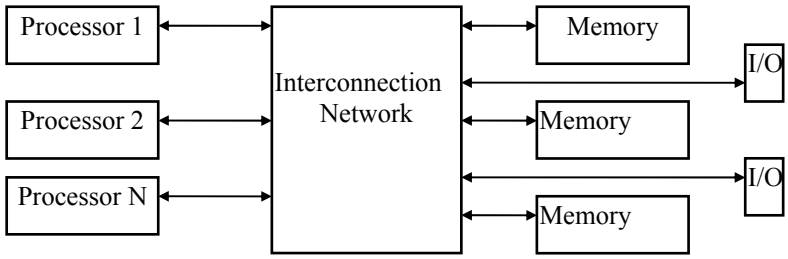


Figure 3.1 (a) :Two multiprocessor structures,
All memories and I/O are remote and shared

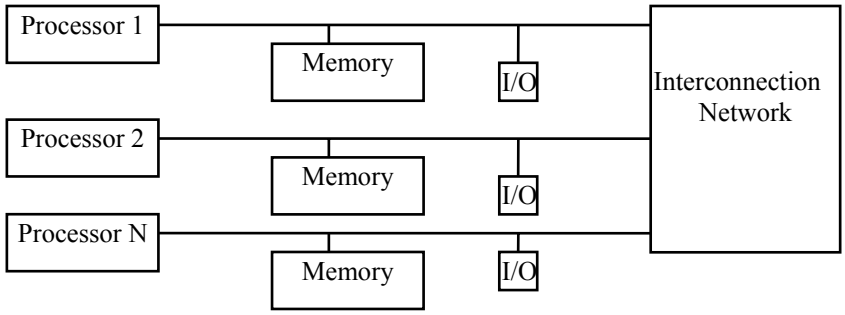


Figure 3.1 (b) :Two multiprocessor structures,
All memories and I/O are local and private

In both systems depicted in figure 3.1 the processors cooperate by exchanging data through the interconnection system and by synchronizing activities. The shared memory in figure 3.1(a) provides a convenient means for information interchange and ynchronization since any pair of processors can communicate through a shared location. The structure in figure 3.1(b) supports communication through point-to-point exchange of information. Obviously, multiprocessors can have any reasonable combination of shared global memory or private local memory. Figure 3.1 shows the extremes in the design space, and practical designs lie at the extremes or anywhere in between.

When a multiprocessor is operating at peak performance, all processors are engaged in useful work. No processor is idle, and no processor is executing an instruction that would not be executed if the same algorithm were executing on a single processor. In this state of peak performance, all N processors of a multiprocessor are contributing to effective performance, and the processing rate increased by a factor of N.

Peak performance is a very special state that is rarely achievable. There are several factors that introduce inefficiency. Among the factors are:

- The delays introduced by interprocessor communications;
- The overhead in synchronizing the work of one processor with another;
- Lost efficiency when one or more processors run out of tasks;
- Lost efficiency due to wasted effort by one or more processors; and
- The processing costs for controlling the system and scheduling operations.

The architect who designs and builds a multiprocessor must pay close attention to the sources of inefficiency exposed here. They can lead to serious degradation in performance. For example, if the combined inefficiencies produce an effective processing rate of only 10 percent of the peak rate, then ten processors are required in a multiprocessor system just to do the work of a single processor.

Multiprocessor Interconnections

- Bus interconnections
- Ring interconnections
- Crossbar interconnections
- Two-and three-Dimensional Meshes
- The Shuffle-Exchange interconnection and the Combining switch
- The Butterfly Operation and the reverse-Binary transformation
- The Combining Network and Fetch-and-Add
- The Hypercube interconnections

The Cosmic Cube (MIMD machine)

A significant difference between the cosmic cube [Seitz 85] and most other parallel processors is that this MIMD machine uses message passing instead of shared variables for

communication between processes. this computational model is reflected in the hardware structure and operating system and is also the explicit communication and synchronization primitive seen by the programmer. The hardware structure of a message-passing machine like the Cosmic Cube differs from shared memory multiprocessor in that it employs no switching network between processors and storage. The hardware structure of the Cosmic Cube, is suitable for highly regular computing problems. The resident operating system of the Cosmic Cube creates a flexible and machine-independent environment for concurrent computations. This process model of computation is quite similar to the hardware structure of the Cosmic Cube but is usefully abstracted from it. The programmer can formulate problems in terms of processes and 'virtual' communication channels between processes.

3.1.1.4 Distributed memory multicomputers

The INMOS transputer [Inmos85] is a VLSI microprocessor designed to be used in distributed memory multicomputers. Transputers can be connected using the communication links to form multicomputers, for example the Meiko Computing Surface [meiko89a]. In common with other transputer based systems the Computing Surface can be configured with electronic wiring chips. These allow users to construct topologies best suited to their current application.

Programming multicomputers like the Meiko Computing Surface has proved to be complex [Meiko89]. The application has to be decomposed into suitable independent tasks and a number of system level issues have to be addressed. These include building communications software to through route messages via intermediate nodes to destination nodes. This can be difficult as deadlock must be avoided and complex data structures converted into packets that can be handled by the routing software. The relatively low bandwidth of transputer links and the direct relationship between distance and latency inherent in store and forward message routing also make network topology an important

factor in building efficient routing software. Application programmers inevitably become involved in a number of non-trivial system level problems.

In order to reduce the complexity of application programming we can view the parallel machine as an abstract machine. This is illustrated in figure 3.2. These abstractions are possible with several available packages and provides communication routing, memory access and process synchronisations. Cstools [Meiko89b] , Tiny [Inmos85], and UBIK [Inmos92] are packages which view the distributed memory multicomputers as a logically fully connected connected networks. This presents the programmer with the communications abstraction write-to-destination. A message is communicated by specifying its destination alone, the routine system is then responsible for its safe delivery. Each processor has the capability to run several concurrent processes and messages can be exchanged both locally and globally.

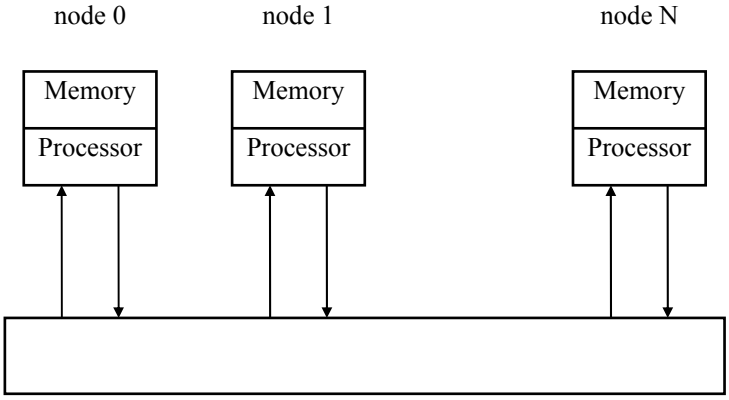


Figure 3.2: Abstract machine architecture

Athas and Seitz [Seitz85] propose a similar programming model for distributed memory multicomputers. They identify an emerging second generation of multicomputers with greater compute performance and much greater message passing performance. One important characteristic of such systems is hardware support for message routing, increasing bandwidth and considerably decreasing latency. Athas and Seitz predict that message latency

will become independent of distance travelled, reducing the topology constraints on application programmers.

The second generation Intel hypercube, the iPSC/2 [Hockney 88], begins to confirm these predictions. Benchmarks [Margulus90] show that latency is becoming less dependent on distance travelled particularly for messages larger than one Kbytes. Another example is the iWarp chip []. The integer processor delivers 20MIPS (peak) while the on chip floating point co-processor has a performance of 20 Mflops (peak). Each iWarp has support for through routing in hardware, with four bi-directional links capable of 80 Mbytes/second. The DOOM machine [] and the Ncube/2 take a similar approach providing hardware support for message routing on partially connected networks.

An alternative is to separate communications from processing and provide an independent communications network. The Nectar system [] is a high performance communications network based on fibre optic technology. This is intended to connect heterogeneous systems of computers together but is also a multicomputer in its own right. Nectarine [], the programming interface to Nectar, will support the write-todestination abstraction.

Adopting the abstract machine architecture as a programming model for distributed memory multicomputers should have several advantages. The write-to-destination abstraction simplifies the programmers task, no longer requiring application programmers to have knowledge of routine algorithms. The abstract machine architecture also provides a level for portability between multicomputers, hiding machine differences beneath a simple abstraction. Finally, this abstraction appears to be realistically based on emerging technology.

3.1.2 Transputers and related systems

3.1.2.1 Transputer architecture & support of concurrency

The transputer [Inmos85] is a single-chip microprocessor from the British company INMOS Ltd. It is distinguished from other vendor's microprocessors in that it is designed as a building block for parallel processors. To facilitate this it has memory and links to connect one transputer to another, all on a single VLSI chip. It can be viewed therefore as a family of programmable VLSI components that support concurrency.

In the transputer INMOS have taken as many components of a traditional von Neumann computer as possible and implemented them on a single 1 cm2 chip, while at the same time providing a high level of support for a concurrent or process-based view of computation.

The support for concurrency includes hardware support for a process queue, with special registers and microcode instructions supporting the creation of parallel tasks; a minimal context for each active process, so that process switching is very rapid; hardware timers; and an external interrupt or event signal.

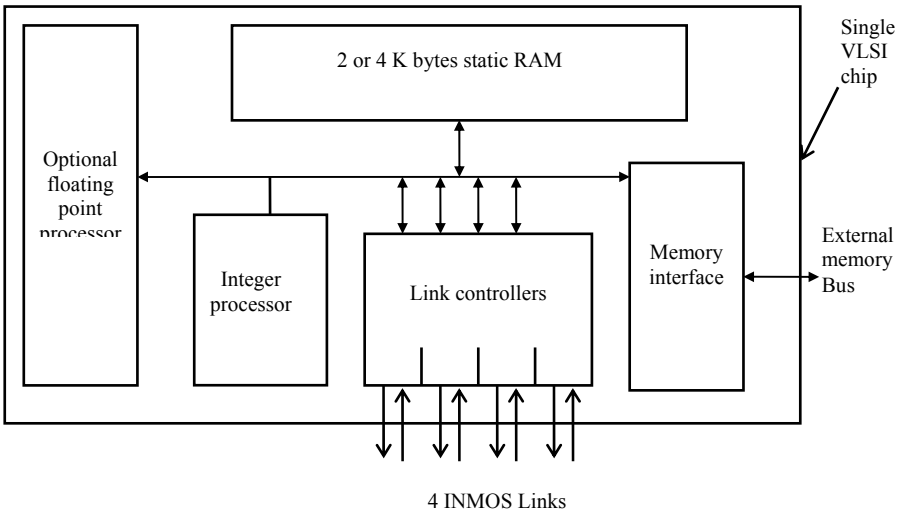


Figure 3.3: The INMOS transputer architecture

The processing part of the transputer is a RISC or reduced instruction set computer, which provides a very rapid rate of execution (20 Mhz) of a small range of instructions and addressing modes.

Transputers products available include a 32-bit processor with a multiplexed 32-bit address/data bus (T414), a 16-bit processor with 16-bit address and data buses (T212), and a disc controller chip, with 16-bit address and data buses and a specialised interface for industry standard discs. A floating point transputer version (T800) has on the same chip a floating-point hardware. Later transputers are the (T9000) which support the UBIK interface [inmos92].

In transputer parts, all components execute concurrently; each of the four links and the floating-point coprocessor on the T800 can all perform useful work while the processor is executing other instructions. Each link has a DMA channel into the memory system which will reduce, but not significantly so, the memory bandwidth to the processor.

3.1.2.2 Support for concurrency

Transputer systems execute the OCCAM programming language , in which concurrency may be described between transputers in the system, or indeed within a single transputer. The transputer must therefore support internal concurrency. A process on the transputer is described by six registers see figure 3.4 The six registers are:

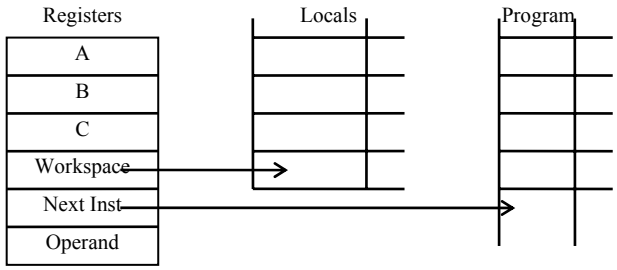


Figure 3.4: The transputer registers .

The transputer has a high degree of support for concurrency. It has a microcoded scheduler, which allows any number of processes to compete for the processor's time, subject to memory limitations only. Processes are held on two process queues. One is the active queue, which holds the process being executed and any other processes waiting on an input, output of timer interrupt. The scheduler does not need to poll any of these devices so consumes no processor cycles on inactive processes. Figure 3.5 illustrates the linkages involved in a process queue, in this case the active queue. Two registers hold the head and tail of the list, and the executing process has its workspace pointer and program counter loaded into processor registers. All other processes hold these values in the workspace.

Process swap times are very small, of the order of microseconds, depending on the instruction being executed. This is because little state has to be saved. The evaluation stack does not need to be saved, and when the current process can no longer proceed, its program counter and workspace pointer are saved in its workspace and the next process is taken from the active list.

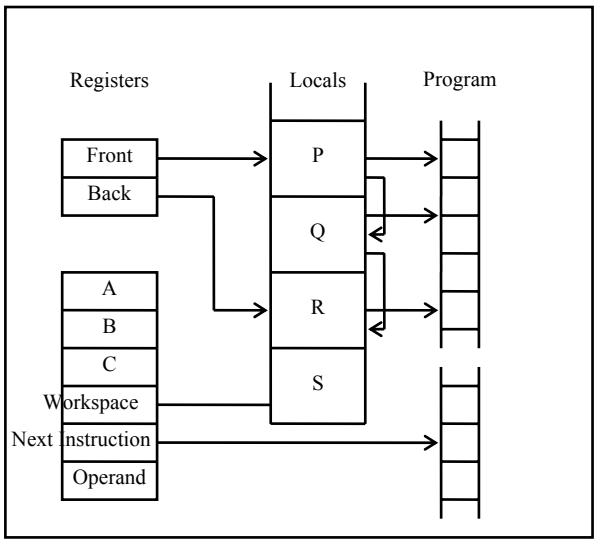


Figure 3.5: The transputer process queue linkage

Two further micro-instructions provide a means of adding and deleting processes from the active process queue. These are *start process* and *end process* . A start process instruction is required for each component of the PAR constructor in OCCAM, and the correct termination of this process is ensured by the use of the end process instruction, which decrements a workspace counter when executed. Only when each component of the PAR construct has terminated will the counter be decremented to zero, signifying correct termination and allowing the process in which the construct was embedded to proceed.

3.1.2.3 The INMOS link

Communication between processes on the transputer or between processes on different transputers is performed by two instructions *input message* and *output message*. The communication which is supported is a point to point , unbuffered message-passing scheme. It therefore requires a handshake between processes, which synchronises them. The same two instructions are used for message passing between processes on the same transputer, as well as between processes running on different transputers. The instructions use the address of the channel to determine which form of communication is required.

Internal channels are represented by a single word in the memory which provides the handshake protocol between communicating processes. The channel holds either a special value ‘empty’ or the identity of a process. A channel is initialised to empty, and when either a reading or writing process becomes ready, the identity of that process is stored in the channel location; This process would then be descheduled. When the second process becomes ready, it will find the process identification of the first process in the channel location and the message is copied and the waiting process is added to the active process list. The message is defined by a count, a channel location, and a pointer to the message. Figure 3.6 shows the above sequence diagrammatically.

External channels proceed in a similar manner, but the link interface performs the job of copying the message across the link circuit. Each link implements two OCCAM channels (or ANSIC channels) in opposite directions over a three-wire TTL level circuit. Communications over these links are controlled by autonomous controllers, which have DMA access to the transputer's memory. Four bi-directional communications and processing can therefore proceed concurrently on the same transputer chip. Each link controller has three registers, holding a pointer to a process workspace, a pointer to a message and a byte count for the message, with which it controls the transfer of the message. The only difference in operation is that on external communication, both processes would need to be descheduled while the autonomous transfer took place.

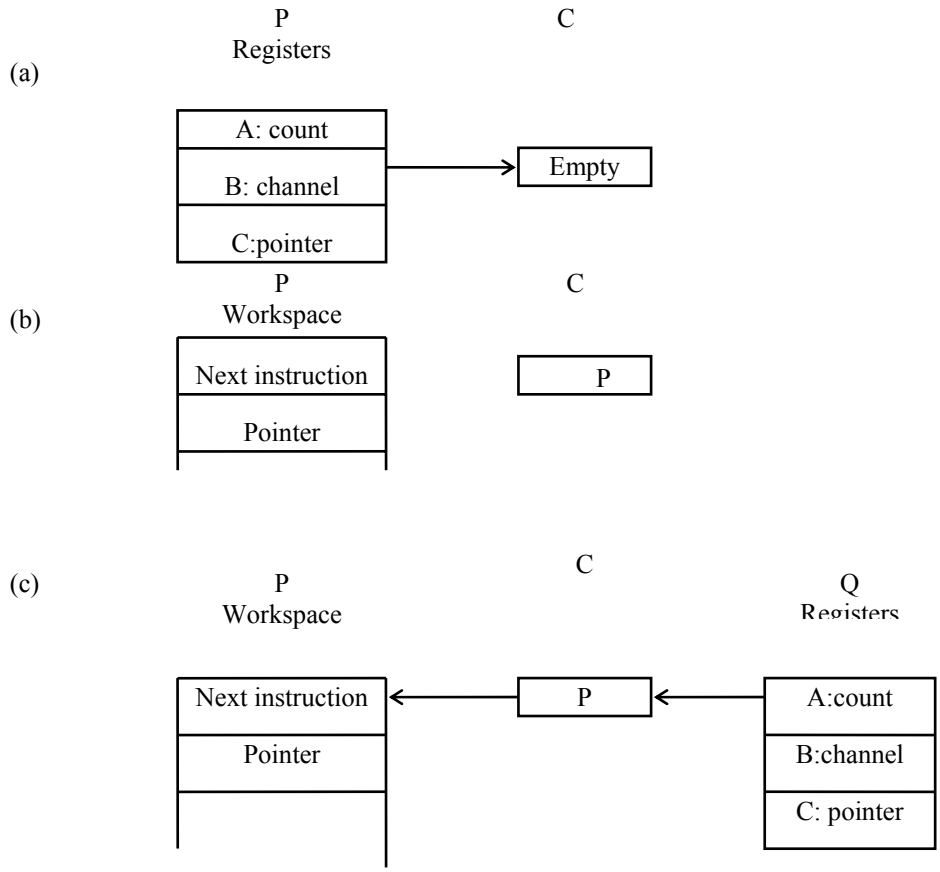


Figure 3.6: The operation of internal and external channel communication.
(a) The first process P finds the channel location empty. (b) The process is descheduled and its location stored in the channel. © The second process Q finds the location of P in the channel, the communication proceeds and process P is rescheduled.

The operation and performance of the INMOS link is fundamental to the effective exploitation of the transputer. All transputer products will support communications over this asynchronous point-to-point connection, with speeds of 5, 10 and 20 Mbit/s. Although the transfer is autonomous, unless the messages passed over the links are reasonably large, a true overlap will not be possible, as each transfer will use a small but finite amount of processor resource to deschedule the processes and initiate set-up of the link controller's registers. If the link bandwidth is measured as a function of message length, the classical pipeline model is seen, with a start-up without transmission, followed by a constant throughput. The granularity of a message therefore modifies the bandwidth seen. It must also be remembered that the start-up period also degrades processor performance, to such an extent that many concurrent small transfers may saturate the processor.

3.1.2.4 Performance

The performance of the transputer is dependent on a number of factors, for example, the clock speed of the part, which may vary from 12.5 MHZ to 20MHZ. Also, if data is coming from off-chip RAM, the access times will depend on the speed of the memory parts used and, for most operations, this is a major factor in the speed of operation. At best, the external memory interface will cycle at three transputer clock cycles (150 ns for the 20 MHZ processor). Internal memory, on the other hand, will cycle within a single processor cycle. It is clear from this that on-chip data will provide significant speed gains.

When using OCCAM the programmer is encouraged to express parallelism, even if the code fragment is configured onto a single transputer. This style of programming should be encouraged, but only providing that parallel processes can be created without excessive

overhead. INMOS claim that in the transputer this overhead is no larger than a conventional function call in a sequential language.

3.1.2.5 Building systems with transputers

Transputers can be used in a variety of applications and in a variety of ways, and there are now many transputer-based systems on the market, including racks of transputer development cards from INMOS [Inmos90]. In this thesis we have used the The B004 and the B008 boards. The B004 board is referred to as the SMT101 Sprint Board, a B004 standard transputer board for IBMPC, from Sundance multiprocessor Technology Ltd [Ainsworth91]. The B008 board is a multiprocessor system in which we can connect the TRAM transputer through a softwire configuration program the MMS (refer to appendix B for more details). However, it does not matter whether code or data may be partitioned over the transputer network; it is the communications issues which determine the mapping of code and the configuration of links.

Communications play an important role in all well designed systems. The communications bottleneck in a conventional processor is typically the processor-to-memory interface, which is usually the off-chip interface. Communications, not surpassingly, can also limit the performance of parallel machines, although here the limit arises in processor-to-processor communications. Moreover, this problem is fundamental, and although it can be moved up and down the system' hierarchy, it will not go away. For example, we can trade chip and wiring complexity in a fully switched system against the large diameter of a static network.

Chapter 3b

3b Performance measures, models of parallel computations, parallel languages

- 3.2 Performance measures
  - 3.2.1 Introduction (Amdahl's Law and other laws)
  - 3.2.2 Definition and Laws
    - 3.2.2.1 Grosch's Law
    - 3.2.2.2 Performance and technology
    - 3.2.2.3 Von Neumann's Bottleneck
    - 3.2.2.4 Amdahl's law
    - 3.2.2.5 The Gustaffson-Barsis Law
    - 3.2.2.6 Fine grain vs. coarse grain machines and R/C ratio
  - 3.2.3 Performance models
    - 3.2.3.1 Basic model: Two processors with .....
    - 3.2.3.2 Extension to N processors
- 3.3 Models of parallel computations
  - 3.3.1 Introduction
  - 3.3.2 Techniques for exploiting parallelism
    - 3.3.2.1 The processor farm
    - 3.3.2.2 Algorithmic parallelism
    - 3.3.2.3 Geometric parallelism
- 3.4 Parallel languages
  - 3.4.1 Introduction
  - 3.4.2 ANSIC toolset
    - 3.4.2.1 Introduction
    - 3.4.2.2 Support for earlier toolsets
    - 3.4.2.3 ANSI C toolset
      - 3.4.2.3.1 ANSI C compiler
      - 3.4.2.3.2 Generating executable code
      - 3.4.2.3.3 loading and running programs
      - 3.4.2.3.4 Program development and support
      - 3.4.2.3.5 Runtime library
      - 3.4.2.3.6 Environment variables
  - 3.4.3 Parallel processing with ANSI C
    - 3.4.3.1 Process, Channel, and Semaphore data types
    - 3.4.3.2 Concurrency functions
    - 3.4.3.3 Processes
    - 3.4.3.4 Channel communication
    - 3.4.3.5 Parallel programming examples

3b Performance measures, models of parallel computations

3.2 performance measures

3.2.1 Introduction (Amdahl's Law and other laws)

Parallel computing has always had its sceptics. First, Grosch's Law had to be overturned by making computers four times as fast in order to sell them for two times as much. Grosch's law was repealed by large-scale integration of electronics.

Then, von Neumann's bottleneck had to be dealt with. A von Neumann computer is limited in performance by the narrow connection between the processor and its memory. The most obvious route around von Neumann's bottleneck is to use parallel processors. but, Amdahl's Law predicted very limited improvement in performance because the speed of a computer was limited by its slowest (sequential) part. Regardless of the number of parallel processors, the problem could never be solved faster than the naturally occurring serial part would permit.

Amdahl's Law was shown to be invalid in certain very interesting cases- cases where the problem size could be increased and the regularity of the problem could be used to feed as many parallel processors as the problem needed. Thus, large matrix calculations could grow even larger without sacrificing speed, if more and more processors were "thrown at the problem." The Gustafson-Barsis Law stimulated great interest once again in parallelism.

This brings up the issue of performance measurement. how should we characterize the performance of a parallel computer when in effect, parallel computing redefines traditional measures such as MIPS (Million Instruction Per Second) and MFLOPS (Million Floating Operations Per Second)? A new measure of performance is needed to relate parallel computing to performance.

The most often quoted measure of parallel performance is the *speedup curve*. This is computed by dividing the time to compute a solution to a certain problem using one processor

by the solution time using N processors in parallel. While this is a popular measure, it is also a controversial one. In this chapter, we examine several versions of speedup as well as other measures of parallel computer performance.

•Some performance measures (MIPS and MSYSPS)

... Efficient mechanisms for ensuring the correct execution of programs...

... The travelling - Salesman Problem is a hard (NP-complete) problem ...

... MSYPS: Synchronizations Per Second pronounced ‘sips’...

...The MIPS measure alone suggests a high throughput, but the architecture constraint on MSYPS can prevent the potential MIPS from being realized.

3.2.2 Definitions and Laws

Aside from the limitations of packaging, heat dissipation, and switching speed of semiconductors, fundamentals such as speed of light and size of circuit would seem to be surmountable by adding parallel processors. This, of course, is the ploy behind parallel computing. Is this really the solution? Some sceptics have claimed that there are even more fundamental barriers to speed, which will prohibit increased performance at any cost. In the following, we examine the reasoning behind this scepticism.

3.2.2.1 Grosch’s Law

An often quoted maxim of the 1960s was a ”Law” credited to Herb Grosh, ” To sell a computer for twice as much, it must be four times as fast.” Grosch reasoned that the rapid advancement of technology combined with the reluctance of buyers to consume new hardware forced buyer and seller to place a value on performance according to his rule of thumb. Buyers of expensive mainframes, he conjectured, were not impressed by marginal performance improvements. They preferred to wait for a fourfold improvement to upgrade the old machine to a new one. On the other hand, sellers placed a premium on speed, and doubling the price of a machine that ran four times as fast seemed like a fair price.

3.2.2.2 Performance and technology

Seymour Cray outsmarted Grosch's Law (for a time) by being uniquely clever. The Cray-1, announced in 1976, was capable of 80 million operations per second (MIPS) and literally defined the word *supercomputer*. By 1988, the Cray Y-MP had crept through the *gigaflop barrier*. To computer engineers, crashing through the gigaflop barrier was psychologically the equivalent of Chuck Yeager breaking the sound barrier.

This speed was largely achieved by using fast circuits and pipelined parallelism. But to be even faster, circuits must be as small as possible, because shrinking reduces the distance information must travel, and hence its transit time. Even with reduced dimensions, supercomputers must use the fastest possible circuits, e.g., gallium arsenide (which switches three times as fast as silicon) and clock at nanosecond rates. (A ray of light can travel about one meter in three nanoseconds.)

With large-scale integration we can construct small, fast computers that are also less expensive to mass produce. The economic law of Grosch is repealed, and it is possible to construct faster and less expensive computers over time.

Ultimately, switching speeds reach a limit and the size of the circuits cannot be reduced so they are limited by the speed of light. The practical performance of a single computer is limited to the current technology. Therefore, parallel computing will be a necessary design alternative, so we must determine the limits to the parallel approach.

3.2.2.3 Von Neumann's Bottleneck

A von Neumann machine consists of a single control unit connecting a memory to a processing unit. Instructions and data are fetched one at a time from the memory and fed to the processing unit under control of the control unit. The processing speed of the entire machine is limited by the rate at which instructions and data can be transferred from memory

to processing unit. This ”narrow” connection between instructions and data held in memory and the single processing unit forms von Neumann’s bottleneck.

The bottleneck can be avoided by removing the assumptions implicit in a von Neumann design. First, we can employ many processing units and many memories. We might also increase the number of control units so that many instruction and data streams are active at once. Finally, we could connect these memories and processors together by some sort of *interconnection network*. Such machines are called *non-von*, because they do not follow von Neumann design.

3.2.2.4 Amdahl’s law

In 1967, Gene Amdahl, an IBM designer said:

> *For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit co-operative solution... The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor... At any time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome.*

Known as Amdahl’s Law, this advice has been quantified into a formula by a number of people who have used it as the major reason why parallel computing cannot defy Grosch’s Law. The following is one such derivation.

Suppose we construct a parallel computer from N processors (vector computer). The purpose of this non-von machine is to run a program consisting of naturally occurring parallel and serial parts.

The diagram in figure 3.7 shows both the advantage of parallelism and the degradation in performance due to serial computations embedded in the program. The use of many processors provides a potential for high performance, but the serial operations reduce the actual effective parallelism below the maximum attainable. To obtain a reasonable measure of

performance for the situation depicted in figure 3.7 , one interesting parameter to study is the *efficiency* of a computation. This is a measure of the proportion of time that the processors are busy. Therefore it measures the degradation in peak performance due to serial operations and other effects.

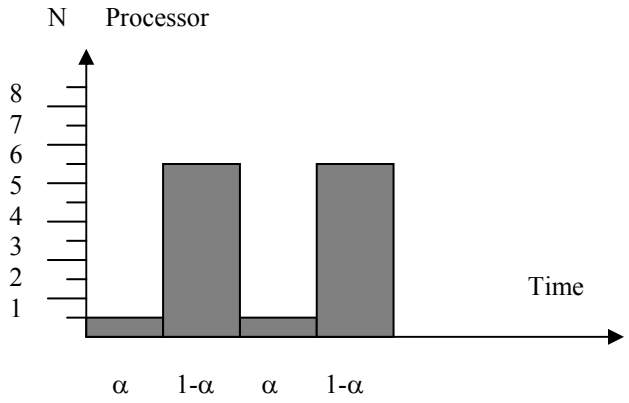


Figure 3.7 : The time history of a computation for an array of processors.

In figure 3.7, let the fraction of time spent in serial code be α and the fraction spent in parallel code be 1- α. With N processors available, the fraction of time that processors are busy is given by:

$$efficiency = \frac{\alpha}{N} + 1 - \alpha = 1 - \alpha(1 - 1/N) \tag{3.1}$$

In fact α, for any specific problem is a function of N, the number of processors used to solve that problem in parallel. To show this relation, let us normalize the computation times so that the serial execution time takes one unit, and the code that can run in parallel takes time p on a single processor. The parallel time shrinks to p/N when run on N processors. The fraction α is the ratio of serial time to total execution time, and is therefore given by

$$\alpha = \frac{1}{1 + p/N} = \frac{N}{N + p} \tag{3.2}$$

The bad news is that for very large N, α approaches 1, so efficiency approaches 0. What is happening is that the parallelizable portions of code take a vanishingly small fraction of time, and what remains is serial, occupying one processor while the remaining processors are idle.

$$efficiency = 1 - \frac{N(1 - 1/N)}{N + p} = 1 - \frac{N - 1}{N + p} \tag{3.3}$$

Consider, for example, p=10,which shows the efficiency for programs for which serial code accounts for only one-tenth as much time as the parallel code when executed on a serial processor. At N=10 the efficiency is only 0.55 and at N=100 the efficiency drops to roughly 0.10.

Another way of measuring efficiency is in terms of speedup, which is essentially a way of expressing how much faster a parallel or pipelined computation is than a serial execution. Let S be the *speedup* achieved by using N processors instead of one processor to solve the problem. We define S as follows:

$$S=T(1)/T(N) \tag{3.4}$$

where T(j) is the time taken to solve the problem using j processors.

The serial part of the program can be computed in time equal to $\beta T(1)$, and the parallel part of the program in time $(1-\beta)T(1)/N$, because we assume the ideal case of "N workers can do the job in fraction 1/N of the time of one worker." That is, $(1-\beta)$ th of the program can be done by N processors in less time than one processor. Then,

$$T(N) = T(1)\beta + \frac{T(1)(1 - \beta)}{N} \tag{3.5}$$

and by substitution into the equation for S,

$$S = \frac{1}{\beta + \frac{(1 - \beta)}{N}} = \frac{N}{\beta N + (1 - \beta)} \qquad \text{(Amdahl's Law)} \tag{3.6}$$

Suppose a program consists of β=0.67 fractional parts of serial code, and 0.33 fractional parts of parallel code. What is the expected speedup for this program when it is run on N=10 parallel processors?

S = 1 / [0.67 + (0.33/10)] = 1.42

Amdahl's Law is very pessimistic because it predicts at best a 50% improvement over a single processor using 10 processors. The economics payoff is simply not there for this problem. Amdahl's Law prevented designers from exploiting parallelism for many years because even the most trivially parallel programs contain a small amount of natural serial code. At β = 10%, for example, Amdahl's Law predicts at best a tenfold speedup. This is hardly good news for designers.

3.2.2.5 The Gustaffson-Barsis Law

In 1988 a team of researchers at Sindia Labs won the *Gordon Bell Prize* for parallel processing with a 1,024 processor nCUBE/10. Seemingly, they overthrew Amdahl's Law by achieving a thousandfold speedup on a problem with β in the range of 0.004 to 0.008. Amdahl's Law predicts speedups ranging from 125 to 250 instead of 1,000. It seemed as if Amdahl's Law had finally been repealed. What did these pioneers do?

John Gustafson and Ed Barsis, wo of the Sandia researchers who shared in the award-winning breakthrough, derived an alternate to Amdahl's Law to explain how they had won. The key is in observing that β and N are not independent of one another. Gustafson states:

The expression and graph (for Amdahl's Law) both contain the implicit assumption that (1-β) is independent of N, which is virtually never the case.

Again, starting with the speedup formula, S = T(1)/T(N), Gustafson-Barsis interpreted this law to mean that parallelism can be used to increase the (parallel) size of the problem. That is, if one processor is used, it must compute both the serial part and the parallel part

$$T(1) = \beta + (1 - \beta) N \tag{3.7}$$

But, if N parallel processors are used, the problem can be scaled up so all N parallel processors execute the serial and parallel part of the program, one followed by the other. Adding the serial and parallel part yields unity as shown below.

$$T(N) = \beta + (1- \beta) = 1 \quad (3.8)$$

Substituting into the speedup formula gives the Gustafson-Barsis Law:

$$S = N - (N - 1)\,\beta \quad \text{(Gustafson-Barsis law)} \quad (3.9)$$

This formula was derived in exactly the same way as Amdahl's law, except T(N) is set to one, meaning the problem is scaled up to fit the parallel computer. Contrary to Amdahl's derivation, T(1) is the time to compute both serial and parallel fractional parts of the program on a single processor.

Keep in mind that these formulas ignore communication costs and overhead associated with operating system functions such as process creation, memory management, and message buffering.

If we calculate the expected speedup for the previous problem using the Gustafson-Barsis Law. Recall N = 10 and $\beta = 0.67$. We find

$S = 10- (9)(0.67) = 3.97.$

Thus, Gustafson_Barsis predicts over twice the speedup of Amdahl's Law. Does this make sense? A fourfold increase in speed is achieved using 10 processors. Given that 67% of the program contained naturally serial code, this is an encouraging result.

3.2.2.6 Fine grain vs. coarse grain machines and R/C ratio

The Connection Machine (CM-1) is an SIMD machine whose 1-bit processors are better suited to fine-grained tasks... The GF-11 is a coarse-grained machine...

What reasoning led the architects of one machine to seek such a vastly different solution than did the architects of the other machine? The range of applications is the primary motivation for the difference. The connection machine is designed to exploit parallelism of tasks such as

image analysis, in which a significant portion of work is characterized by fine-grained tasks. The GF-11, which is designed for much larger-grained tasks,..... Thus the architects of each machine attempted to match granularity to applications for the machine.

Cray XMP – a four-processor system in which each processor is a Cray I supercomputer... Communication occur only at the end of major phases. Smaller granularity is evident on microprocessor-based-multiprocessors such as the Cosmic [Seitz85] Cube and a number of commercial versions of this hyper cube based design.

The point of this section is to analyse the performance benefit of multiprocessors in the face of overhead incurred to create parallelism.

The performance benefits strongly depend on the ratio **R/C** , where R is the length of a run-time quantum and C is the length of communications overhead produced by that quantum. The ratio expresses how much overhead is incurred per unit of computation. When the ratio is very low, it becomes unprofitable to use parallelism. When the ratio is very high, parallelism is potentially profitable. Note that a large ratio can be obtained by partitioning a computing job into relatively few large pieces, and that the amount of parallelism for such a ratio might be much smaller than the maximum available.

The ratio R/C is a measure of task granularity:

- In coarse grain parallelism, R/C is relatively high, so each unit of computation produces a relatively small amount of communication; and
- In fine-grain parallelism, R/C is very low, so there is a relatively, large amount of communication and other overhead per unit of computation.

The programmer seeking maximum performance is strongly tempted to partition a problem into the finest possible granularity to create the maximum amount of parallelism. But if the maximum parallelism also has the maximum overhead, it is not clear that maximum parallelism leads to the fastest solution.

The remainder of this section is devoted to performance models. In each model, we observe how the ratio R/C determines the strategy that achieves the optimum performance.

3.2.3 Performance models

3.2.3.1 Basic model: Two-processors with unoverlapped communications

Consider an application program that contains M tasks. Our objective is to execute this program at maximum speed on a system with N processors. For simplicity, we first consider a system with just two processors and then let the number of processors increase. To model performance we need to characterize the combination of execution time and overhead that will be incurred.

Let us make the following assumptions to obtain our initial results. Subsequently we relax the assumptions and see how the performance changes. Specifically, we assume that:

1. Each task executes in R units of time; and
2. Each task communicates with every other task at an overhead cost of C units of time when the communicating tasks are not on the same processor, and at no cost when the communicating tasks are coresident.

Although we use the notation C as if C were exclusively due to communication, it is convenient to lump overhead from all sources into C. The total processing time is the following:

$$\text{Execution Time} = R \, \text{Max} (M-k , k) + C (M-k) k \qquad (3.10)$$

The run time for two processors is the larger of the run times experienced and is therefore the larger of R(M-k) or Rk when k tasks are assigned to one processor and M-k to the other. Note that the first term is a linear function of k ; and the second term is a quadratic function of k.

What is the minimum execution time for equation 3.10 as a function of k? That is , how shall we assign tasks to two processors to produce the minimum execution time? Figure 3.8 shows a graphical way of finding a solution. The answer for this model is to assign all

tasks to one processor if R/C is below M/2, or split the tasks evenly between two processors if R/C exceeds that threshold. That is, either k=0 or k=M/2.

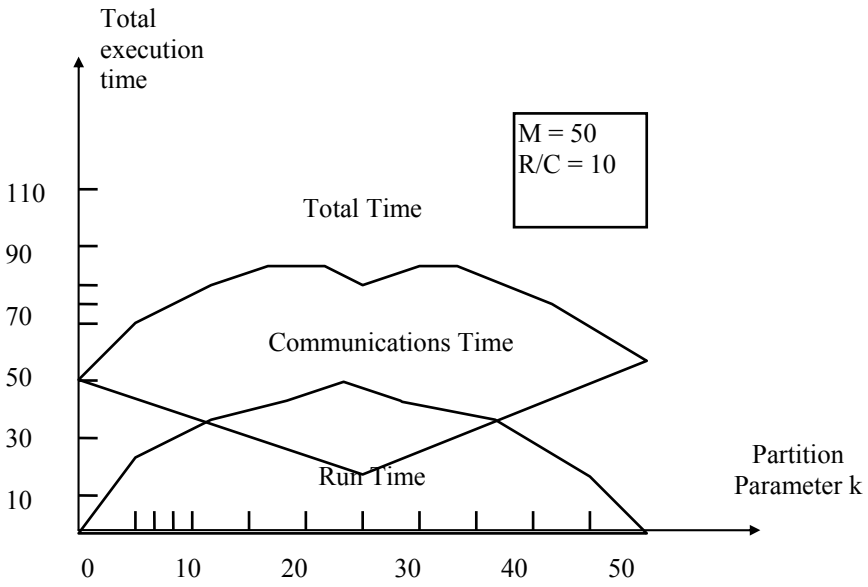


Figure 3.8 (a): Parallel execution time for R/C = 10.
Optimum partition parameter k=0.

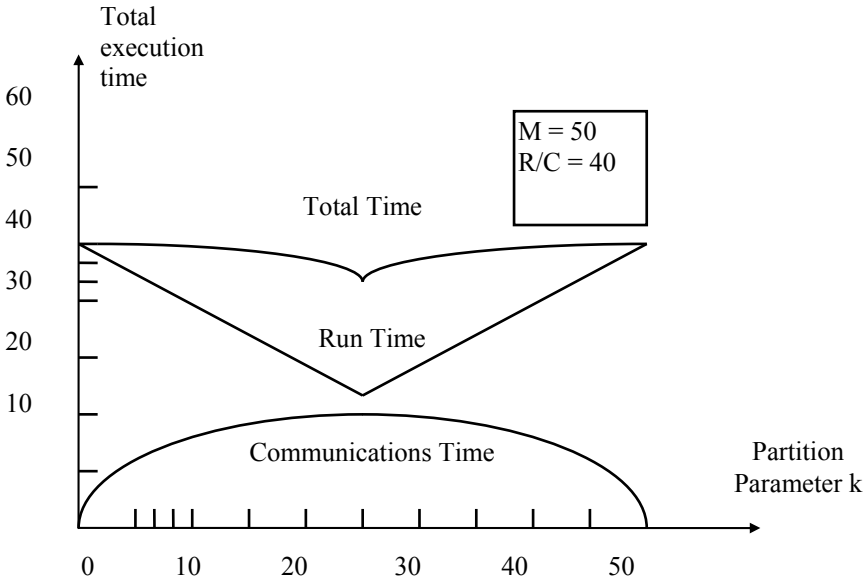


Figure 3.8 (b): Parallel execution time for R/C = 40.
Optimum partition parameter k=M/2.

3.2.3.2 Extension to N processors

Let us consider what happens when there are N processors. In this case we assign k_i tasks to the ith processor. The generalization of equation 1.5 becomes :

$$Execution - Time = RMax(k_i) + \frac{C}{2}\sum_i k_i\,(M - k_i) = RMax(k_i) + (\frac{C}{2})(M^2 - \sum_i {k_i}^2) \tag{3.11}$$

The first term counts the longest running time among the N execution times. To that time is added the overhead from the second term. That term counts the number of distinct pair-wise links between k_i tasks and M - k_i tasks, each of which contributes an amount C to the total time. The second term in Eq.(3.11) is quadratic just as in Eq.(3.10).

The difference in cost of the 'even' distribution to N processors and a 1-processor assignment is given by

$$Time - Difference = \frac{RM}{N} + \frac{CM^2}{2} - \frac{CM^2}{2N} - RM \tag{3.12}$$

Where the first three terms form the cost of the even distribution of tasks and the last term is the cost of assigning all tasks to one processor.

To simplify the analysis, we have ignored values of M that are not exact multiples of N. To solve for the threshold value of R/C, we set the value of Eq.(3.12) to 0. By removing a factor of M and then grouping terms by coefficients R and C, we can remove another factor of (1-1/N). This yields the equation

$$R/C = M/2 \tag{3.13}$$

This model shows that if R/C is greater than the threshold M/2, then an even distribution of tasks to as many processors as are available will produce the best time. On the other hand, if R/C is below that threshold, then no matter how many processors are available, no assignment

produces a faster time than the assignment that uses only one processor. here is a situation in which the role of overhead becomes quite clear.

Although this analysis has looked at performance rather than costs, R/C determines the point at which parallelism is cost effective. Even when R/C is sufficiently high to warrant parallelism, the performance gain is diminished by the second term of Eq.(3.11). The speedup attributable to parallelism is the ratio of time to run on one processor to time expressed by Eq.(3.11). This is approximately

$$Speedup = \frac{RM}{(\frac{RM}{N} + \frac{CM^2}{2} - \frac{CM^2}{2N})} = \frac{\frac{RN}{C}}{(\frac{R}{C} + \frac{M(N-1)}{2})} \tag{3.14}$$

If the first term of the denominator is large compared with the second, then the speedup is proportional to N. This requires M and N to be small and for R/C to be large. If parallelism is increased to the extent that the denominator is dominated by its second term because N is very large, the speedup is proportional to R/CM, which does not depend on the number of processors. Hence, as N increases, the speedup approaches a constant symptote.

At this point each processor added to the system brings extra cost while yielding negligible performance benefit. Even though performance can improve incrementally as processors are added, the diminishing returns in performance are not worth the added cost. The number of processors should not be increased beyond some maximum that is a function of cost and the ratio R/C.

This model is a general picture of how granularity and overhead affect the performance gain of a multiprocessor, and it gives some indication of the importance of minimizing overhead and selecting the right granularity. It is only one model, however, and it cannot encompass the full spectrum of the actual applications.

Let us alter the model in various ways and observe how the findings change. In general, we discover that R/C plays a critical role, regardless of the model. In some cases, there is the same type of threshold in which the best solutions are extreme. That is, use all available processors or just one processor, depending on the value of R/C. In some models, the extreme solutions are n of the best. The best solutions for these models distribute work among several processors,, but do not use all processors because the use of too many leads to performance degradation and extra cost. Moreover, in the general case work need no be distributed evenly to achieve the optimum performance.

To summarize the findings of the models presented in this section, we have discovered:

1- Multiprocessor architecture produces an overhead cost that is an additional burden not present in serial processors and vector architectures. The overhead cost includes the cost of scheduling, contention for shared resources, synchronization, and processor-to-processor communications.

2-Although running time for a computational portion of a program tends to diminish as the number of processors working on that program increases, the overhead costs tend to grow with the number of processors. In fact, it is possible for overhead costs to grow faster than linearly in the number of processors.

3- The ratio R/C is a measure of the amount of program execution (running time) per unit overhead (communication time), within a program implementation on a specific architecture. The larger this ratio, the more efficient the computation because a relatively smaller proportion of time is devoted to overhead as this ratio increases. However, if the ratio is made large by partitioning a computation into a few pieces instead of many small pieces, the parallelism available is greatly reduced, which limits the speedup that can be attained on a multiprocessor.

We clearly have a dilemma. On the one hand, R/C has to be small to create a large number of potentially concurrent tasks, and on the other hand, R/C has to be large to prevent the overhead costs from becoming excessive. Because of the dilemma, we cannot expect to build fast multiprocessors simply by expanding the number of processors as much as technology allows.

There is some maximum number of processors that is cost-effective, and that number depends a great deal on the architecture of the machine, on the underlying technology (especially communications technology), and on the characteristics of each specific application.

3.3 Models of parallel computation

3.3.1 Introduction

When setting out to design a parallel algorithm one must ask two questions of the problem at hand:

(1) Can enough parallelism in the problem be identified to allow efficient solution?

(2) Can the processors share the data necessary in the problem fast enough (given their organisation) to allow an efficient solution?

The first question is the most fundamental, in that problems with little inherent parallelism will never have good parallel solutions and is largely independent of what parallel computational model one chooses to use.

On the other hand, the answer to the second question depends on the model used.

Models of parallel computation can be broken roughly into two groups. Special-purpose models and general-purpose models. The later models are those where powerful and general communications are assumed, typically in the form of a large synchronized shared memory accessible by all processors. Such assumptions allow researchers to focus on the fundamental characteristics of a parallel computation, and ignore the issues which arise through particular choices in architecture and interconnection networks. More specifically, general-purpose models allow one to consider problem 1 above, while not being distracted (annoyed) by the compounded difficulty of solving problem 2 simultaneously. In addition to its simplicity, the use of such an abstraction is further justified by the rate at which parallel architectures change in practice, which causes results regarding special-purpose models to have limited relevance over time. the most common general-purpose model is the Parallel Random Access Machine, or PRAM. An (n,m)-PRAM consists of n processors and m memory locations, where each processor is a random-access machine. It is a synchronous model, in that no processor will

proceed with instruction i+1 until all have finished instruction i. Within this synchronous restriction a PRAM may execute in SIMD mode or in MIMD mode.

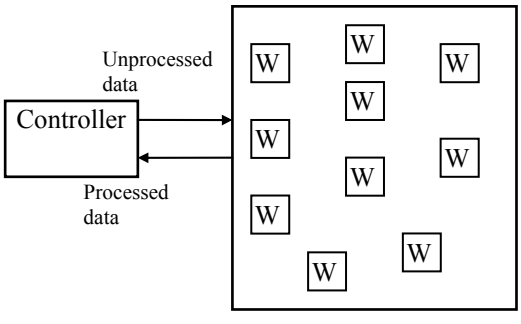
3.3.2 Techniques for exploiting parallelism

The techniques which can be applied to the solution of a problem is dependent on many factors, including the languages available, the underlying model of parallelism, and the parameters of its implementation. However, the same problem may be implemented in many cases using all three of the techniques outlined below.

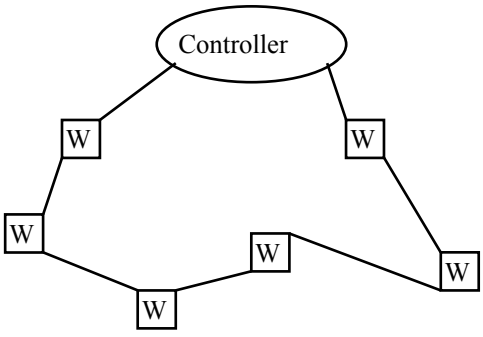
3.3.2.1 The processor farm

This is one of the simplest methods of exploiting parallelism. It can be used in applications where discrete bundles of processing can be 'farmed out' to an available processor. These processes may well be identical, but must be independent.

Process structures for implementing processor farms are illustrated in figure 3.9(a). Figure 3.9(b) shows a single-level model, where work is distributed along a linear structure when requested, or even continuously, with results being collected along back channels. Each process, which would be physically distributed, would contain code for receiving and forwarding work (which would depend on the buffering scheme used),and a copy program to be executed.



3.9(a): Process farm



3.9(b) :single-level model linear structure

3.3.2.2 Algorithmic parallelism

The second technique relates to pipeline structures. In this technique, a program or process is partitioned and distributed across a network of processors, or processing agents . Each partition executes in parallel on its own data ,or data that it receives form other partitions of the program . Where there are sequential constraints on the evaluation of the partitions of the program, pipelining may be brought info play to maintain concurrency. Where pipelining is used, there is a requirement for streams of data to be processed by each partition, so that any start-up imposed by sequence may be amortised.

The network of processors required for this technique must ideally reflect the data flow of the given algorithm. It should in fact form a static data flow graph of the algorithm. The granularity of the distributed processes could theoretically be at the level of a single operation, as in fine grain dynamic data flow machine, or even finer if we consider the microprogramming of pipelined architectures. Alternatively, the granularity could be at the program level, as in UNIX piped processes. The granularity of the processes must be determined by the ratio of time required for processing, compared with the time required for communicating any data to a given process.

3.3.2.3 Geometric parallelism

The final technique for exploiting replication involves a partitioning of data, rather than code. If a given algorithm can process a large data structure concurrently, then it may be partitioned over a network of processors. In this case ,the algorithm normally used to process that data would simply be replicated on every processor in the network . The algorithm would then be applied to the local partition of the data structure, together with any data communicated form other partitions.

3.4 Parallel languages

3.4.1 Introduction

There are two fundamental techniques available to explicitly express parallelism; structure parallelism and process parallelism. Various implementations of the first of these, like CMLISP, and even APL, which provide mechanism which treat operators as the basic values of the parallel data structure. We could, of course, imagine whole programs as being the components of the parallel structure, in which case we have a description of process parallelism. To reiterate then, the distinction is one of granularity, for structure parallelism was defined as the granularity of a single operation over element of a data structure. Process parallelism, however, requires distributed sequences of operations.

The underlying computational model and the manner in which load is balanced across a system is also very different between these two approaches. In structure parallelism one can consider the processors to be associated one per data structure element, with activated data being mapped onto the available processors in order to balance the load. Thus, load sharing is achieved through data structure element redistribution. In process parallelism, however, the process is virtualised and load balancing occurs by the distribution of processes across the processors. The unit of distribution is the code, and of course its associated data and state. The virtualisation in this case involves maintaining a number of instruction streams on a single processor.

3.4.2 ANSI C toolset

3.4.2.1 Introduction

The ANSI C tools has been designed to reflect the parallel processing model of communicating sequential (CSP). The inherent flexibility of the C language the capacity to mix form different languages, and the ability to use the concurrency features make ANSI C a powerful tool for programming concurrent systems.

3.4.2.2 Support for earlier toolsets

A file converter tool supplied with the toolset enables object code and libraries generated by earlier INMOS compilers and toolsets such as the 3L Parallel C and occam 2 toolsets to be incorporated into programs written with ANSI C. Specific support is provided for functions from the 3L Parallel C toolset.

3.4.2.3 ANSIC Toolset

3.4.2.3.1 ANSI C compiler

The compiler icc is an ANSI standard C compiler with additional support for concurrency. It conforms fully with ANSI standard X3.159 1989.

The ANSI standard for C formalises the original implementation of C described in 'The C programming Language' by Kernighan and Ritchie, and extends it to include a runtime library, some language extensions already in common usage, and many other improvements designed to standardise the language.

ANSI C supports concurrency through a series of C structures and a comprehensive set of process handling, channel communication, and semaphore manipulation functions. Some useful non-ANSI functions are also provided in the runtime library.

3.4.2.3.2 Generating executable code

Three tools are used in sequence (or two for single transputer program) to generate the loadable file from compiled object code:

ilink - the toolset linker which links separately compiled program units
icconf - the configurer tool which generates a configuration data file (for multitransputer programs only)
icollect - the code collector which generates a bootable file for a transputer network either from the configuration data file or a single linked unit.

3.4.2.3.3 Loading and running programs

Bootable code for single transputers and transputer networks is loaded onto the transputer hardware using the host file server tool ***iserver*** which both loads the program and

starts up the runtime environment that supports interaction with the host. The auxiliary skip loading tool ***iskip*** can be used in combination with ***iserver*** to load a program onto an external network.

3.4.2.3.4 Program development and support

Seven tools are provided to assist in program development:

idebug - the interactive network debugger.
idump - the memory dump tool for use with idebug when debugging programs on the root transputer.
ilibr - the librarian which generates libraries of compiled code.
ilist - the binary lister which decodes and display data from object files.
isim - the T425 transputer simulator.
imakef - the Makefile generator which creates Makefiles for toolset object files.
icvlink - the file format convertor which allows object code to be imported from earlier INMOS toolsets.

Figure 3.10 illustrates the development process in terms of the architecture of the toolset. The default file extensions assumed and generated by the tools are used to represent source and target files.

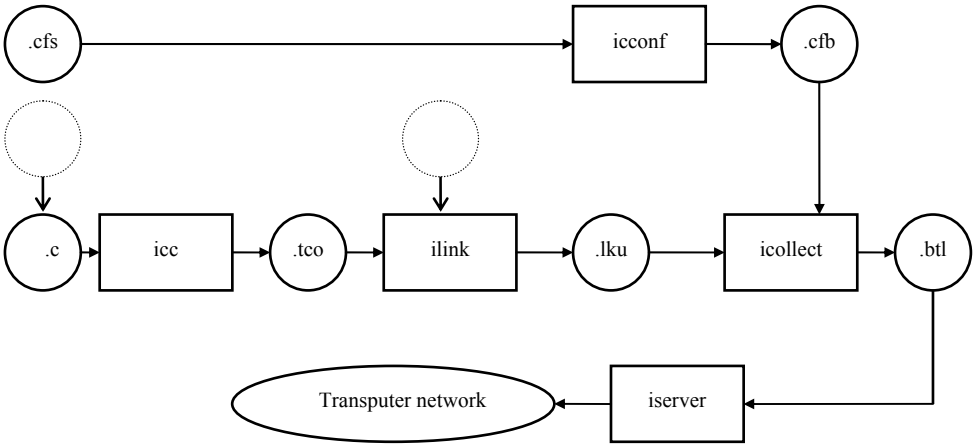


Figure 3.10: Toolset compilation architecture

3.4.2.3.5 Runtime library

The runtime library is a library of compiled C functions that perform common programming operations. The library contains the complete set of ANSI standard functions plus functions to support parallel programming and some non-ANSI extensions. The concurrency functions are

divided into three functional groups: process management, channel communication handling, and semaphore handling. The non-ANSI extensions include a set of i/o primitives, a set of short maths functions, functions for retrieving information about the host system, and debugging functions.

3.4.2.3.6 Environment variables

The toolset uses a number of environment variables on the host system. Use of the variables is optional but if defined they will affect the behaviour of the tools on your system table 3.3.

Table 3.3: Ictools environment variables

Variable	Meaning
ISEARCH	The search path i.e. the list of the directories that will be searched if a full pathname is not specified.
ITERM	The file that defines terminal keyboard and screen control codes. used by idebug.
IBOARDSIZE	The size (in bytes) of memory on the transputer board. Used by iserver.
TRANSPUTER	The address at which the transputer board is connected to the host. Used by iserver.
IDEBUGSIZE	The size (in bytes) of memory connected to the root transputer. Used by idebug.
toolname ARG	Default command argument.

3.4.3 Parallel processing with ANSI C

3.4.3.1 Process, Channel, and Semaphore data types

Three new data types complete the concurrency support. Data structure are used to hold data about processes and semaphores, and a pointer type is used to implement channels.

- **Process**: A structure type that holds information about each declared process.

- **Channel**: A pointer type used to implement channels. In accordance with the CSP model, channel variables represent unidirectional communication links between two processes. Channel is a pointer to type void.

- **Semaphore**: A structure type that holds information about a semaphore.

Parallel processes are created by linking a function definition to a predeclared process structure, and are then initialised, started, and run using routines from the concurrency library. Channels between processes are created simply by declaring a variable of type Channel * at an appropriate point in the program. Channel input and output functions are then used to pass data. It is the responsibility of the programmer to ensure that data sent by one process is received by another; separate functions exit for input and output and the two must be paired for communication between two processes to take place.

Semaphores are declared using either the semaphore initialisation function or a macro that performs a similar action. Semaphores are then acquired and released by calls to two separate functions. Semaphores can be used to synchronise the activity of low with high priority processes.

3.4.3.2 Concurrency functions

The concurrency functions implement the following parallel processing operations:

- Process setup, startup, and scheduling
- Ready input selection
- Channel communication
- Semaphores.

The main parallel processing functions are declared in the header files *process.h* and *channel.h*. Declarations of functions for semaphore handling can be found in *semaphor.h*.

3.4.3.3 Processes

Processes are defined in the same way as regular C functions, but with fixed first parameter. The first parameter to a new function which will be started as a process must be a pointer to its own ***Process*** structure. Parameters to the function follow the fixed process pointer in the normal way.

Example of the creation and instantiation of a process is shown in figure 3.11

```
void newproc(Process *p, int arg1, int arg2, int arg3)
{
        p=p;
        /* ... process code */
}
int main()
{
        /*   Declare pointer to process structure */
        process *x;
        /*   Declare parameters *:
        int pa1, pa2, pa3;
        /*   Allocate process; check for non-allocation */
        if ((x = ProcAlloc(newproc,0,3,pa1,pa2,pa3)) == NULL)
                abort();
        /*   Start process running */
        ProcRun(x);
        /*   Rest of code executes
                in parallel with 'newproc' */
}
```

figure 3.11: Example of the creation and instantiation of a process

• Process initialisation

Two functions allocate and initialise parallel processes. A third function is provided to allow parameters to be altered in an existing process. The three functions and their parameters are listed below in table 3.1.

Table 3.1: Functions to allocate and initialize processes.

Function	Parameters
Process ProcAlloc	(void (*func) (), int size, int nparam,...)
int ProcInit	(Process *p, void (*func) () , int *ws, int wssize, int nparam, ...)
void ProcParam	(Process *p, ...)

• Freeing the stack and workspace

ProcAllocClean (Process *p) is used for processes initialised using ProcAlloc, and ProcInitClean (Process *p) is used for processes inititalized with ProcInit.

• Process execution

A set of functions is provided for executing processes asynchronously (ProcRun and related functions) or synchronously (ProcPar and others). Functions are provided in the first group to start processes at high or low priority, and in the second group to start many processes in a single call or to start a pair of processes at high and low priority.

Table 3.2: Functions to execute processes asynchronously.

Function	Parameters
void ProcRun	(Process *p)
void ProcRunHigh	(Process *p)
void ProcRunLow	(Process *p)
void ProcPar	(Process *p1, Process *p2, ...)
void ProcParList	(Process **plist)
void ProcPriPar	(Process *phigh, Process *plow)

• **Unsynchronised processes**

Unsynchronised processes (those started with the ProcRun, ProcRunHigh, and ProcRunLow functions) run independently of the main program and may continue when the main program terminates. To ensure that processes do not access the server when it has already been terminated, processes can be coupled to the main program using channels. The main program will then wait until all other processes are finished before it terminates the server. Alternatively, ProcPar can be used to force synchrony an a group of processes. An example of how to use synchronisation channels is shown in figure 3.12.

```
#include <process.h>
#include <channel.h>
void p1(Process *p, Channel *synch)
{
  p = p;
  /* ... process code */
  ChanOutInt(synch,1);
/* Send integer to main program to signal completion*/
}
int main()
{
Process *p;
Channel *synch; /* Synchronisation channel*/
if ((synch = ChanAlloc()) == NULL
      { /* Call channel error handler */ }
if ((p = ProcAlloc(p1, 0, 1, synch))  == NULL)
      { /* Call process error handler*/ }
ProcRun (p);
ChanInInt(synch);
/* Receives completion signal from process p1*/
} /* Program can terminate safely */
```

Figure 3.12: Unsynchronised processes.

• Process timing and scheduling

Routines are provided for delayed execution, the timed suspension and rescheduling of processes, and the termination of process before normal completion. Table 3.4 lists the timing and scheduling routines.

ictools routines for delayed execution	
Function	Parameters
void ProcAfter	(int time)
void ProcWait	(int time)
void ProcReschedule	(void)
int ProcGetPriority	(void)
void ProcStop	(void)
int ProcTime	(const int time1, const int time2)
int ProcTimePlus	(const int time1, const int time2)
int ProcTimeMinus	(const int time1, const int time2)
int ProcTimeAfter	

• **Input alternation**

Six routines are provided (see table 3.6) to allow for the selection of a ready channel from multiple parallel inputs. Separate versions of the routines are provided to deal with lists of channels.

• **Simple alternation**

ProcAlt and ProcAltList suspend the current process until one of the channel arguments is ready to input. On the completion, the functions return an index into the parameter list indicating the ready channel.

For example, the following code sets i to 0, 1, or 2, according to which of the three channels becomes ready first:

```
int i;
Channel *c0, *c1, *c3;

i = ProcAlt (c0, c1, c2, NULL);
```

Both ***ProcAlt*** and ***ProcAltList*** require at least one input parameter; if the parameter list is empty an error is generated.

• **Polling several inputs**

ProcSkipAlt and ***ProcSkipAltList*** check a series of channels without blocking the current process. If one of the channels is ready to input an index into the parameter list is returned. Both functions return immediately with a special code value and do not wait for a channel to become ready.

ProcTimerAlt and ***ProcTimerAltList*** block the current process until one of the channels is ready for input or until a specified time is reached. If a channel becomes ready before the timeout occurs an index into the parameter list is returned, otherwise the timeout code is returned.

3.4.3.4 Channel communication

Routines are provided for the passing of bytes and integers on channels (***ChanIn***, ***ChanOut***, and others), for implementing safe channel protocols (extraordinary link handling), and for allocating and resetting channels.

Communication between processes is effected by passing data on variables of type ***Channel*** . Functions are provided to allocate , initialise , and reset channels, to support input and output of characters, integers, and untyped data, and to assist with establishing reliable protocols (extraordinary link handling).

Channel input and output functions must be paired for two processes to communicate and exchange data.

• **Transputer link addresses**

Each link on a transputer is associated with an input and an output channel address. Transputer link addresses are defined in the library header file **channel.h**.

• **Channel allocation, initialisation , and reset**

ChanAlloc reserves space from the heap for the channel, initialises the channel and returns a pointer to it. If space cannot be allocated ***ChaAlloc*** returns NULL.

chariest resets a channel to its quiescent (non-communicating) state, returning either a descriptor to the process waiting to communicate, or the value ***NotProcess_p*** which indicates

the previous communication completed successfully and the channel is free. **NotProcess_p** is a macro defined in the header file channel.h.

•Channel Input and Output

ChanOut and ***ChanIn*** perform the basic operation of passing bytes on a channel. The data can be of any type or size but the number of bytes must be specified. Typed data should be broken down into individual bytes for transmission and retyped on input.

ChanOutChar and ***ChanInChar*** are similar except that they pass single characters and no byte count is required. ***ChanOutInt*** and ***ChanInInt*** are similar except that they pass single integers. The six functions are listed in table 3.6

Table 3.6:Channel Input and Output

Function	Parameters
void ChanOut	(Channel *c, void *cp, int cnt)
void ChanIn	(Channel *c, void *cp, int cnt)
void ChanOutChar	(Channel *c, char ch)
void ChanInChar	(Channel *c)
void ChanOutInt	(Channel *c, int n)
void ChanInInt	(Channel *c)

• Reliable channel protocols

Four functions are provided to allow recovery from link failure. The functions are essentially the same as ChanOut and Chanin except that the communicating process becomes rescheduled after a specified timeout peiord or after receiving a communication on a special reset channel. The four functions are listed in table 3.7.

Table 3.7: Reliable channel protocols

Function	Parameters
int ChaOutTimeFail	(Channel *chan, void *cp, int cnt, int time)
int ChanOutChanFail	(Channel *chan, void *cp, int cnt, Channel *failchan)
int ChanInTimefaill	(Channel *chan, void *cp, int cnt, int time)
int ChanInChanFail	(Channel *chan, void *cp, int cnt, Channel *failchan)

• Semaphores

Semaphore handling routines are provided for programmers who wish to write traditional parallel code based on the acquisition and release of tokens. The routines are used within the implementation of INMOS C to parallelize certain functions in the standard i/o library. The standard process and channel functions provide the best way of using the transputer hardware to execute parallel code. Semaphores are used in the language implementation to parallelize some library routines. For instance, they are used in the implementation of ***malloc***, ***free***, and ***realloc***to prevent the heap being corrupted by simultaneous calls from concurrently executeing processes.

• **Semaphore allocation**

Two functions and a macro are provided to set up and initialise semaphores. All perform the same basic operation of creating a semaphore and their use depends on personal choice see table 3.8.

Table 3.8: Semaphore functions

Function	Parameters
Semaphore *SemAlloc void SemInit SEMAPHOREINIT	(int initvalue) (Semaphore *sem, int initvalue) (int initvalue)
void SemWait void SemSignal	(Semaphore *sem) (Semaphore *sem)

3.4.3.5 Parallel programming examples

The following three examples are parallelized version of the « Hello World » program. They are designed to demeonstrate :

- How to set up parallel processes
- How to bind processes together using a synchronisation channel
- How to couple processes together for the exchange of data

Example1: Unsynchronised parallel processes

This example shows how to declare and run basic parallel processes in C. A time delay is introduced into one of the processes to demonstrate their independence from each other

```
#include <stdio.h>
#include <stdlib;h>
#include <process.h>

void hello(Process *p)
  {
    p = p;
    ProcWait (10000);
    printf(« \Hello,\n »);
  }

void world(Process *p)
  {
    p = p;
    printf(« \nWorld\n »);
  }

int main()
  {

    Process *p1, *p2;

    p1 = ProcAlloc(hello, 0, 0);
    if (p1 == NULL)
      abort();
    p2 = ProcAlloc(world, 0, 0);
    if (p2 === NULL)
        abort();

    ProcPar(p1, p2, NULL);

  }
```

Example 2: Process synchronised by a channel

The example shows how the two processes in example 1 can synchronise their activity using channel communication. Using a channel to connect the two processes forces process world to wait until hello has completed its output, and makes the processes interdependent. No status polling is required because synchronization is implicit in the channel reference.
The integer channel functions are used for convenience. In this instance any pair of channel functions will do the job providing the communication protocols agree. The channel simply ties the two processes together, and communicates no real data.

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>
void hello(Process *p, Channel *ready)
        {
                p = p;
                ProcWait(10000);
                printf('\nHello, \n');
                ChanOutInt(ready, 1);
        }
void world(Process *p, Channel *ready)
        {
                p = p;
                ChanInInt(ready);
                printf('\nWorld\n');
        }
int main()
        {
                Process *p1, *p2;
                Channel *ready;

                ready = ChanAlloc();
                if (ready == NULL)
                        abort();

                p1 = ProcAlloc(Hello, 0, 1, ready);
                if (p1 == NULL)
                        abort();
                p2 = ProcAlloc(world, 0, 1, ready);
                if (p2 == NULL)
                        abort();

                ProcPar(p1, p2, NULL);

        }
```

Figure 3.20: Processes synchronises by a channel

Example 3: Communicating data over a channel

The example shows how two processes can both synchronise their behaviour and communicate data by the use of channels.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <process.h>
#include <channel.h>
void input(Process *p, Channel *chan)
        {
                char message[20];
                p = p;
                printf('\nPlease type your name (20 letters max :');
                gets(message);
                ChanOut(chan, message, 20);
        }
void display(Process *p, Channel *chan)
        {
                char name[20];
                p = p;
                ChanIn(chan, name, 20);
                printf('\nHello %s\n', name);;
        }
int main()
        {
                Process *p1, *p2;
                Channel *chan;

                chan = ChanAlloc();
                if (chan == NULL)
                        abort();

                p1 = ProcAlloc(input, 0, 1, chan);
                if (p1 == NULL)
                        abort();
                p2 = ProcAlloc(display, 0, 1, chan);
                if (p2 == NULL)
                        abort();

                ProcPar(p1, p2, NULL);

        }
```

Figure 3.21: Communicating data over a channel

3.5 Parallel programming using ANSI C toolset on MIMD networks supported by the IMS B008 transputer board and The S708 software support

3.5.1 Introduction to the IMS B008 Board

3.5.1.1 The IMS B008 Transputer board

3.5.1.2 Configuration through switches

3.5.2 The MMS Module Motherboard Software (S708 Device driver)

3.5.2.1 Installing the S708 device driver

3.5.2.2 Using the MMS to run application programs

3.5.2.3 Menu options

3.5.2.4 Description of the softwire configuration

3.5.3.5 Network mapper

3.5.3 Installing the IMS D7214 ANSI C toolset

3.5.3.1 Introduction

3.5.3.2 Installing the release

3.5.3.2.1 Installation

3.5.3.2.2 Setting up the toolset for use

3.5.3.3 confidence testing

3.5.1 Introduction to the IMS B008 Board

3.5.1.1The IMS B008 Transputer board

The IMS B008 is a Transputer Module (TRAM) motherboard designed to plug into a PC or PC/AT bus. The board has ten TRAM slots, an interface to the PC bus and an IMS C004 link switch to allow networks of TRAMs to be set up under software control. Figure 3.3 shows the existing slots on the current board.

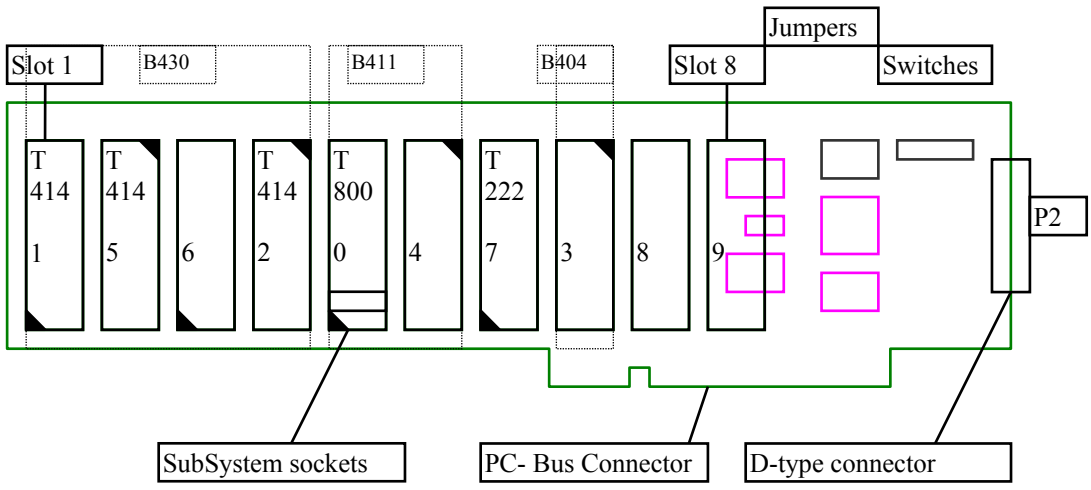


Figure 3.3: Block diagram of the B008 transputer board

3.5.1.2 Configuration through switches

Table 1: Configuration through switches		
switches	state of the switches	configuration
sw1,sw2	sw1:on and sw2:off	PCBus interface enable (default board at address 150hex
sw3	on	System services from host system services
sw4	on	TRAM 1-9 system services from TRAM in slot 0
sw5-sw8	off	20 Mbits/s
JP1&JP2		Link wiring options

3.5.2 The MMS Module Motherboard Software (S708 Device driver)

3.5.2.1 Installing the S708 device driver

Config.sys should have the line:

DEVICE=C:\S708\S708DRIV.SYS /A 150 /D 1 /I 11
DEVICE=C:\ictools\iterms\bansi.sys

Autoexec.bat should include the following lines:

1-PATH ... C:\S708;
2- SET ITERM=C:\S708\PCMMS.ITM

3.5.2.2 Using the MMS to run application programs

iserver /sb mms2.b4 softwire B008

3.5.2.3 Menu options

The menu options available are as follows: H,Q,S,C,T,N,M,L,V,R,I,B,O.

3.5.2.4 Description of the softwire configuration

Softwire:

```
SOFTWIRE
        PIPE 0
                SLOT 1, LINK 3 TO SLOT 5, LINK 0
                SLOT 1, LINK 0 TO SLOT 7, LINK 0
        END
```

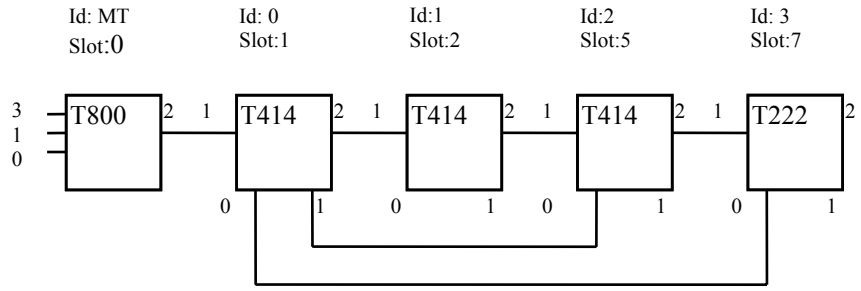


Figure 2.9: Softwire configuration

3.5.2.5 Network mapper

If we set the B008 with the previous softwire configuration and display the mapping through the iserver menu we obtain the following :

Link		0	1	2	3
Id	Type	Id Link	Id Link	Id Link	Id Link
MT*	T800d	----	----	0 * 1	----
0*	T414b	2 * 3	MT * 2	1 * 1	1 * 0
1*	T414b	0 * 3	0 * 2	2 * 1	oooo
2*	T414b	oooo	1 * 2	3 * 1	oooo
3*	T222c	oooo	2 * 2	oooo	oooo

3.5.3 The IMS D7214 ANSI C toolset

3.5.3.1 Introduction

Layout of the ANSI C toolset delivery manual is: Chapter 1: Introduction (Contents & prerequisites), Chapter 2: Installing the release, Chapter 3: Confidence testing, Appendix A: Distribution kit, Appendix B: Debugger function keys ; Shows the positions of the debugger and simulator function keys on the IBM PC keyboard.

Prerequisites

AN IBM PC , PC/XT, or PC/AT (or compatible); DOS version 3.0 or later; About 7Mbytes of free disk space (Although you need not install the entire release); An IMS B004, B008 (or similar) transputer board with an IMS T800 or T414.

Contents of the release

A set of eleven 360 Kbyte 5.25 floppies & five 3.5 floppies; ANSIC toolset delivery manual; ANSIC C toolset user manual; ANSIC toolset reference manual; ANSIC toolset handbook.

PC hosted tools

Two versions of some of the tools are supplied ; Transputer bootable and PC hosted executable. It is up to the user to decide which versions to use.

3.5.3.2 Installing the IMS D7214 ANSIC toolset

3.5.3.2.1 Installation

To install the release first insert disk1 in your floppy disk drive and run the batch file **install.bat** by typing the command **b:install b: c** then insert the diskettes 2 through 5 as they are requested. Answer yes to the question **do you want T225 supports.** At the end of the installation you will receive the message **INSTALLATION COMPLETE.** You may then delete install.bat.

The installation procedure creates a directory called **\ictools**. All the programs necessary to install the toolset are copied to this directory. All the components of the toolset are copied into sub-directories of \ictool, as shown in the following table:

Directory	**Contents**
\ictools\itools	The transputer bootable tools
\ictools\tools	The PC hosted tools
\ictools\libs	The toolset libraries and include files
8ictools\examples	Examples directory
\ictools\examples\simple	Simple example sources
\ictools\examples\debugger	Debugger example sources
\ictools\imakef	Imakef example sources
\ictools\config	Configurer example sources
\ictools\config\b008	Configurer example B008 config files
\ictools\iserver	The iserver executables
\ictools\source	Source code
\ictools\source\iserver	Server sources(see user manual)
\ictools\source\imakef	Imakef sources
\ictools\iterms	Example interm files and drive program
\ictools\nec	NEC PC support (if installed)

3.5.3.2.2 Setting up the toolset for use

This ction explains how to set up the environment necessary to use the toolset

config.sys

-device [high]=c:\ictools\itools\bansi.sys
-shell=command.com /e:1024 /p

autoexec.bat

-path c:\dos; c:\ictools\iserver;c:\ictools\tools;c:\ictools\itools
-set iterm=c:\ ictools\iterms\pcbansi.itm
-set IBOADSIZE=#200000

-set IDEBUGSIZE=#200000

3.5.3.3 Confidence testing

This section describes a short procedure which may be followed to check that installation has been done correctly.

```
1- C>cd \ictools                                                    ;Move to ictools directory
2- C:\ictools> mkdir abbes                                          ;user directory
3- C:\ictools> cd abbes                                             ;i.e. abbes
4- C:\ictools\abbes>copy \c:\ictools\examples\simple\hello.c ; copy hello.c to abbes
5- C:\ictools\abbes> icc hello /ta                                  ;Compile hello.c for TA
6- C:\icttols\abbes > ilink hello.tco /f startup.lnk /ta             ;Link hello.tco with rtl.lib
7- C:\ictools\abbes > icollect hello.lku /t                         ; Collect the code
8- C:\icttols\abbes > iserver /sb hello.btl                         ; Run the executable hello.btl
9- c:\ictools\abbes> isim hello.btl                                 ;Invokes the simulator on hello.btl
```

Chapter 4:

Design of two parallel solutions to the mssm algorithm with a logically fully connected network supporting message routing.

In this chapter we present the design of two parallel solutions to the multiple scale signal matching algorithm (**mssm**) [Mowforth89] over the meiko computing platform which is an MIMD machine composed of INMOS transputer boards and running Cstools [Meiko89]. This Cstools is a message passing system which connects the transputers in a fully connected network and supports message routing.

The first parallel solution is based on decomposing the algorithm into its elementary processes with a pipelined parallel model [Kung88]. This approach of programming takes into account the pipelined nature of the visual channels in the mssm algorithm.

The second parallel solution is based on data partitioning and fine granularity processes. The processes are scheduled according to the process farm model [May87] which insures an automatic load balancing even for heterogeneous networks.

4.1 Parallel solution 1: (Task decomposition & Pipelined model)

This solution is based on decomposing the algorithm into the elementary processes which perform the filtering and matching within one visual channel of the **mssm** algorithm. Each of these elementary processes computes their V successive tasks in a pipelined manner and sends the processed data to the next process linked to it according to the parallel model of figure 4.1. These processes do not block since their computations are transmitted in formatted messages which are queued in the communication channel buffers of the Cstools as show in figure 4.2. However, it is expected that the matching process blocks for the discrepancy map of the previous visual channel before it proceed.

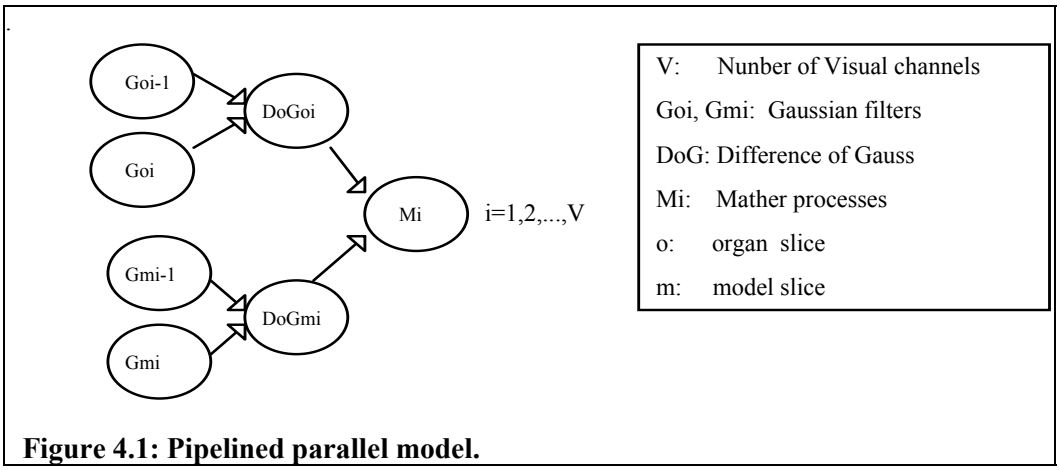


Figure 4.1: Pipelined parallel model.

The parallel model of this solution is explained in detail in the following sections along with the performance evaluation of three different configurations that have been experimented and evaluated with two NMR images and various sets of application parameters represented in the number of visual channels (*V*) and the elasticity parameter (*E*).

4.1.1 The model

The mssm algorithm can be decomposed into seven elementary processes: Two separated filter processes for each image slice of the patient's organ and of the anatomical model ($\mathbf{Go_i}$, $\mathbf{Go_{i-1}}$, $\mathbf{Gm_i}$, $\mathbf{Gm_{i-1}}$), the Difference of the Gaussian of the image slices

(**$DoGo_i$, $DoGm_i$**) , and of the matching processes **M_i**. This is repeated iteratively for the N vision channels of the algorithm according to the parallel model of figure 4.1.

The idea is to design a parallel model where all the processes: Go_i, Go_{i-1}, $DoGo_i$, Gm_i, Fm_{i-1} , $DoGm_i$ are executed in parallel. Then the output of these processes are passed to the matcher process M_i. While the matcher process M_i is matching the filtered image and model with a coarse filter; The processes Go_{i+1}, Go_i, $DoGo_{i+1}$, Gm_{i+1}, Gm_i , $DoGm_{i+1}$ perform the filtering of the model and image with a finer filter.

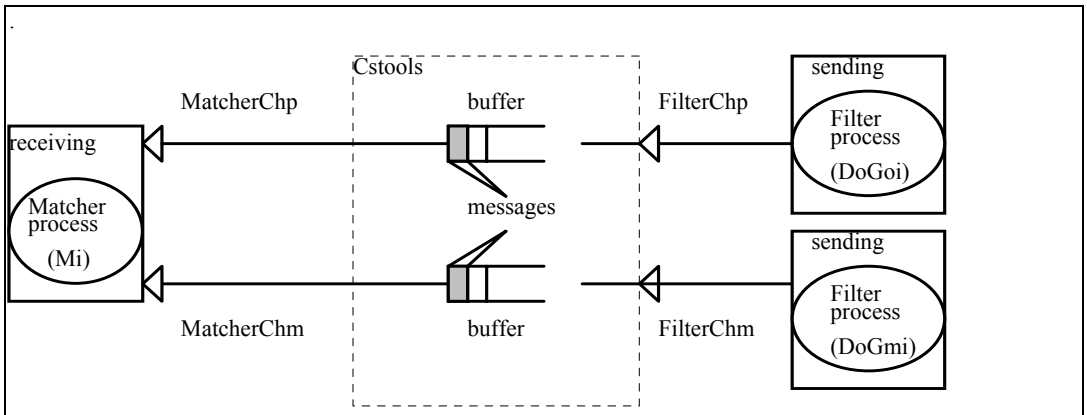


figure 4.2: buffered synchronization between Mi and DoGoi, DoGmi.

All the mentioned processes have a decentralized type of control. Each process at start, reads the task queue from a parameter file *inputfile.dta* (which holds parameters such as the names of the files representing the patient slice image and the anatomical model image, the Elasticity parameter (*E*), and the number of visual channels (*V*) and copies the patient and model images into the processor local memory. The task queues provide the sequence of tasks to be performed by each process. The synchronization between processes is based on a message passing system (MPS) of MEIKO [Meiko89a] known as the Cstools. This message passing system is based on Hoar Sequential Communicating Processes work [Hoar78]. In addition the synchronization between processes is buffered which means that the sender and

receiver processes will maintain the synchronization between transmitted and received messages with a FIFO buffer as shown in figure 4.2. Each message is formatted with one field which hold the data to be communicated between linked processes.

4.1.2 Programming with Cstools

The seven processes (Go_i, Go_{i-1}, Gm_i, Gm_{i-1}, $DoGo_i$, $DoGm_i$, M_i) have been placed on the processors according to three different configurations a, b, and c with 3, 5 and 7 processors respectively as shown in figure 4.3.

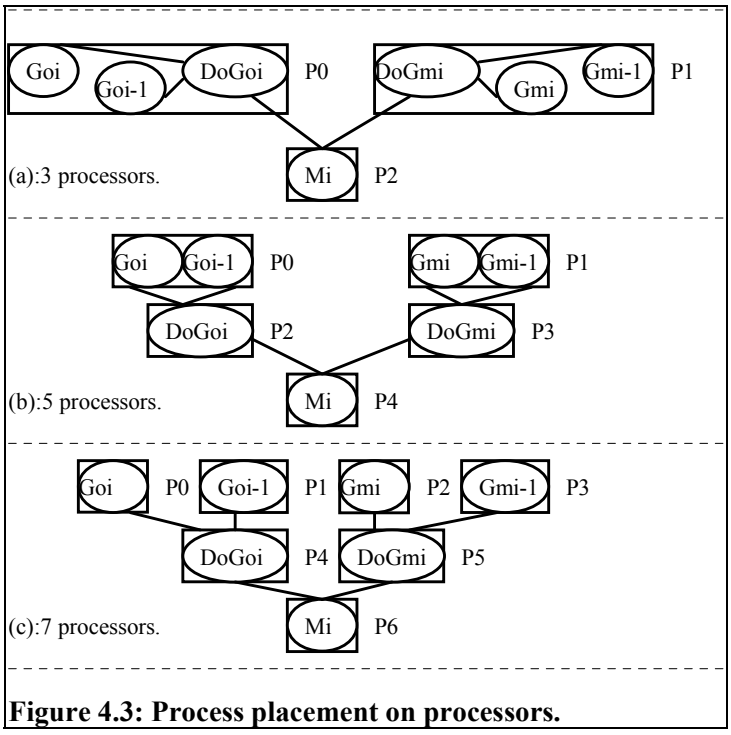


Figure 4.3: Process placement on processors.

In order to show the programming of the this parallel solution we will select configuration (a) and describe the three programs: Filter_Patient_Process(), Filter_Model_Process(), and Matcher_Process(). These programs are described in a pseudo code language in programs 2, 3, and 4. They will be loaded on processor 0, 1, and 2 using a par file of Cstools described in program 1.

```
par
        processor 0 Filter_Patient_Process   inputfile.dta
        processor 1 Filter_Model_Process     inputfile.dta
        processor 2 Matcher_Process          inputfile.dta
endpar
```
program1:Parfile of configuration 4.3(a).

The Cstools interface provide a library of commands to build parallel systems. The procedures Filter_Organ_Process() , Filter_Model_Process() and Matcher_process() listed in program2, program3, and program4 respectivelly show the necessary set of commands to send and receive messages between the Matcher_process (Mi) and the Filter_processes (DoGoi) and (DoGmi) through the connected channels MatcherCho, FilterChp and MatcherChm, FilterChm previously shown in figure 4.2.

```
Filter_Organ_Process(Organ_Slice_Ptr, Filterd_Slice_Ptr1, Filterd_Slice_Ptr2, size)
Char *Organ_Slice_Ptr, *Filtered_Slice_Ptr1, *Filtered_Slice_Ptr2;  int size;
{
 char *t_message_o;
 Transport FilterCho;   netid_t MatcherId;   cs_getinfo(&nProcs,&procNo,&localId);
 cs_open(CSNNULL_ID,&FilterCho);   csn_lookupname(&MatcherId, matchero, 1);
 for(i=0;i<V;i++){
     receive filtered images from Gpi,Gpi-1;
    DoGoi(Filtered_Slice_Ptr1, Filtered_Slice_Ptr2, t_message_o);
    csn_tx(FilterChp,0,MatcherId,*t_message_o, size);
}
}
```
Program2: Filter_Organ_Process (Dogpi) , Refer to figure 4.2.

```
Filter_Model_Process(Model_Slice_Ptr, Filterd_Slice_Ptr1, Filterd_Slice_Ptr2, size)
Char *Model_Slice_Ptr, *Filtered_Slice_Ptr1, *Filtered_Slice_Ptr2;  int size;
{
 char *t_message_m;
 Transport FilterChm;   netid_t MatcherId; cs_getinfo(&nProcs,&procNo,&localId);
 cs_open(CSNNULL_ID,&FilterChm);  csn_lookupname(&MatcherId, matcherm, 1);
 for(i=0;i<V;i++){
     receive filtered images from Gpi,Gpi-1;
    DoGmi(Filtered_Slice_Ptr1, Filtered_Slice_Ptr2, t_message_m);
    csn_tx(FilterChm,0,MatcherId,*t_message_m, size);
}
}
```
Program3: Filter_Model_Process (Dogmi) , Refer to figure 4.2.

```
Matcher_process(r_message_o, r_message_m,  X_Map_Ptr, Y_Map_Ptr, size)
Char *r_message_o, r_message_m, X_Map_Ptr, Y_Map_Ptr;
int size;
{
 char *r_message_o,*r_message_m;
 Transport MatcherCho, MacherChm;  cs_getinfo(&nProcs,&procNo,&localId);
 cs_open(CSN_NULL_ID,&MatcherCho) ; csn_registername(MatcherCho,matchero);
 cs_open(CSN_NULL_ID,&MatcherChm) ; csn_registername(MatcherChm,matcherm);
 for(i=0;i<V;i++){
     csn_rx(MatcherChm,*r_message_m, size);
     csn_rx(MatcherCho,*r_message_o, size);
     Match(r_message_m, r_message_o, X_Map_Ptr,Y_Map_Ptr);
   }
}
```

Program4: Matcher_Process (Mi) ,Refer to figure 4.2.

4.1.3 Performance analysis

The application parameter sets of {*V, E*} have been varied with the values *V* = 8, 4 for vision channels and *E* = 4, 2 for the elasticity parameter. The NMR slices used to measure the performance of this parallel solution have two different image sizes of width x height = {256 x 256 and 128 x 128} pixels corresponding to NMR slices obtained from the Leeds General Infirmary. A 128 x 128 NMR slices representing a cut in the human brain is shown in figure 4.4.

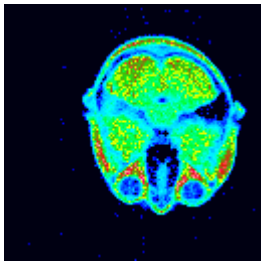


figure 4.4: NMR slice (128 x128 x 256)

Figures 4.5(a),4.5(b), and 4.5(c) present the timing of this parallel solution with the three processor configurations shown in figure 4.3 and the application parameters used to measure the performance in table 4.1 above.

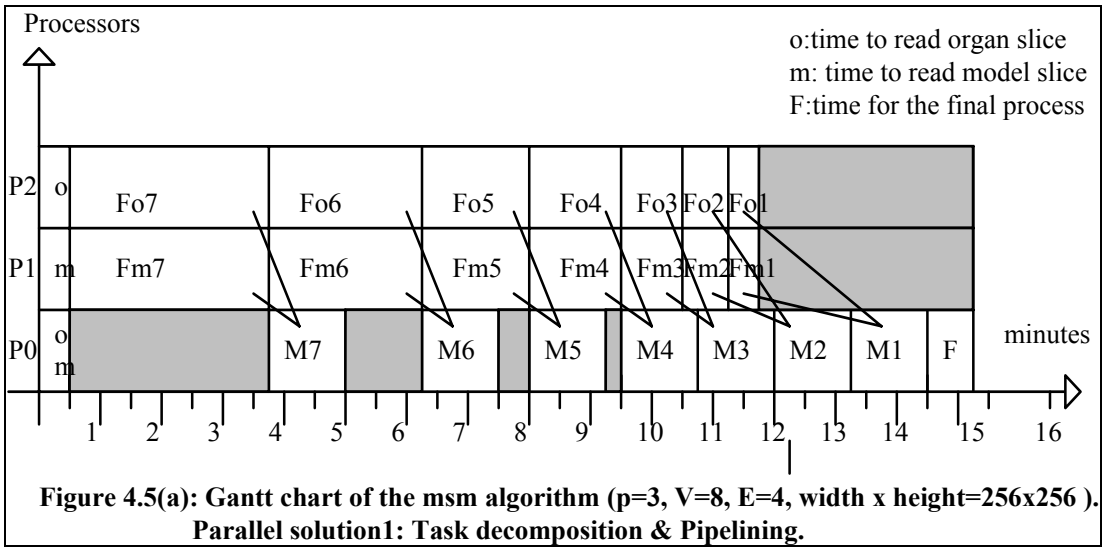


Figure 4.5(a): Gantt chart of the msm algorithm (p=3, V=8, E=4, width x height=256x256). Parallel solution1: Task decomposition & Pipelining.

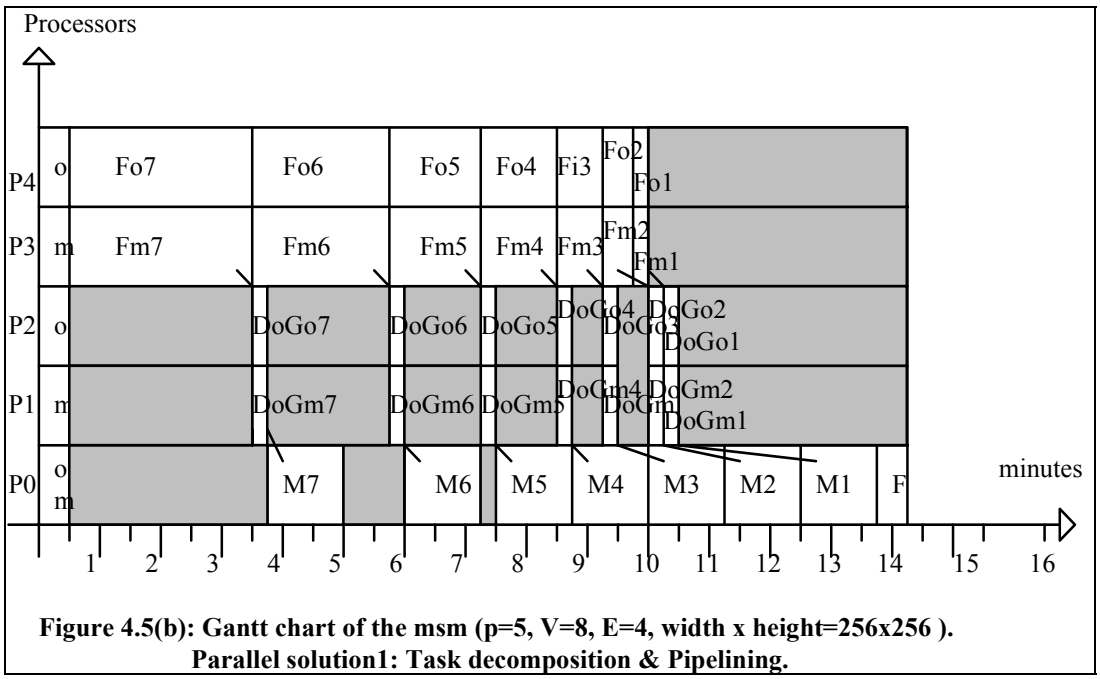


Figure 4.5(b): Gantt chart of the msm (p=5, V=8, E=4, width x height=256x256). Parallel solution1: Task decomposition & Pipelining.

The performance of each configuration have been measured with the speedup S (refer to 3.2.2.2 S=T(1)/T(N)) and processors idle time (Ips/Eps) as show in Table 4.1. Where T(1) and T(N) are the response time of the serial and parallel programs; Ips and Eps are the processors idle times and execution times. The speedup reaches 2.94 with topology three (7 processors). But this gain is obtained with a high processor idle time (Ips/Eps=57%). Which means that the processors spend 50% of their time idling.

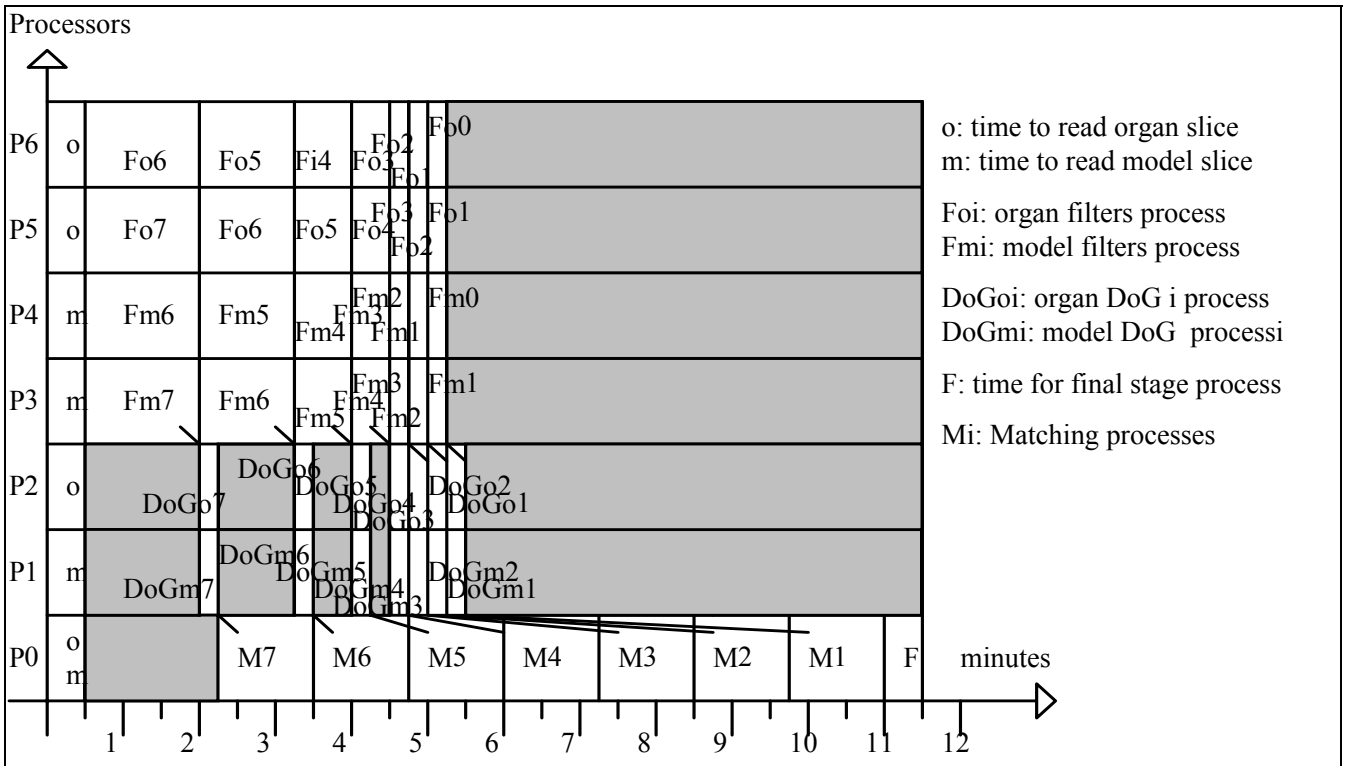


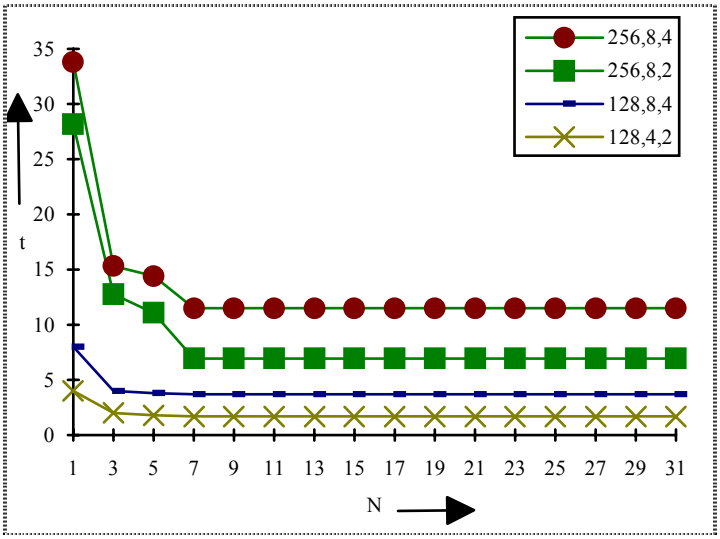
Figure 4.5(c): Gantt chart of the msm algorithm (p=7, V=8, E=4, width x height=256x256).

Parallel solution1: Task decomposition & Pipelining.

Table 4.1: Performance (S=(T(1)/T(N)) (task decomposition & pipeline model) with x height =256 x 256 , T(1)=33.83 m, E=4 pixels, V=8 vision channels.				
Number of processors N	Parallel time Pt	Speedup S= (T(1)/T(N)	Efficiency	Processors usage (idle time/ execution time) (Ips/Eps)
3	15.31	2.20		0.16
5	14.42	2.34		0.50
7	11.50	2.94		0.57

Figure 4.6 is a plot of the execution time versus the number of processors The four curves have been measured with the parameter sets {*size, V, E*} where *size* corresponds to the edge of image, *V* to the number of vision channels, and E to the elasticity parameter . It shows that the model is saturated after 7 processors. Theoretically, it is possible to add more processors and parallelize all the visual channels. However, this will not improve the

performance of the parallel program due to the sequential nature of the matching processes M_0 , M_1 , ... , M_{V-1}.



t: times in minutes, **N:** number of processors.
Fig 4.6: Execution time vs. Nb. of processors.

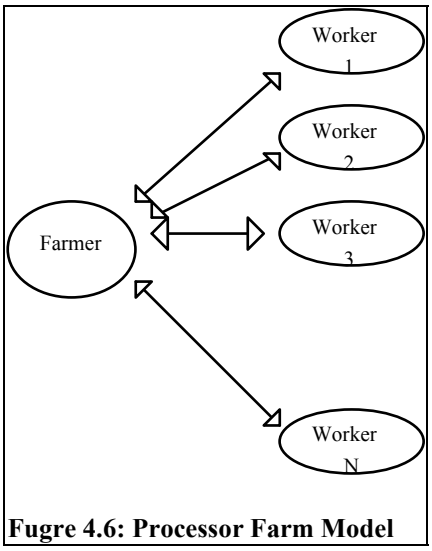
4.1.4 Conclusion

The Gantt charts drawn in figures 4.5(a),4.5(b)and 4.5(c) show in fact that the processors idle time is not negligible, therefore the model does not utilize efficiently the processors. To lessen this idle time , it is possible to work out a better placement policy of the processes over the network topologies[charef94]. Therefore, we can improve in one hand the response time of the model; on the other hand we can obtain a better load balancing.

We conclude this section by saying that it is required from the designer to exploit the technique of data partitioning instead of task decomposition to reach a better response time. The data partitioning technique allow us to have an equal amount of tasks quantified in term of task granularity and if they are scheduled with an efficient parallel model we can reach higher performance and efficiency. This is the topic of the next parallel solution that we will present in the next section.

4.2 Parallel solution 2: (Data partitioning & Process farm).

The design of this parallel solution is based on partitioning the patient and model images into 2D grids of data segments that we call grains (G_t) see figure 4.8. Then we will consider the matching of these patient data segments with their corresponding model data segment as tasks. These tasks are executed by the processors of the network according to the process farm model [May87] which is based on a controller process called **farmer** created in the root processor and a set of similar processes called **workers** replicated in all the nodes of the network. The controller process and the distributed processes form a parallel system connected in a tree network. This is possible through the Cstools buffered channels [Meiko89a, Meiko89b] as shown in figure 4.6 and it is independent of the network topology.



Fugre 4.6: Processor Farm Model

The controller process partitions the organ-slice into a grid of segments of granularity (G) and build a queue of tasks. Then it schedules these tasks for processing by the workers following a demand driven scheme. Data (organ-slices) and Filter operators (Guassian operators) are replicated in the local memory of each processor to reduce the communication overhead through the exploitation of locality of data access.

4.2.1 The process farm model

To simplify the task of developing application software it is often convenient to use a more restrictive programming model. One such model is the processor farm computational model proposed by May and Shepherd [May87]. This model is suitable for a variety of applications, ranging from graphics and image processing algorithms to numerical problems and simulation.

The processor farm computational model has become popular amongst the transputer using community for a variety of applications. A number of features of the model have contributed to its popularity and adaptability. Providing there is sufficient memory in each worker processor to store the data. The model allows an arbitrary number of processors to be used concurrently to solve one or a variety of problems using a single processor acting as controller and one or usually more worker processors connected into a linear topology see figure 4.7. The main task of the controller (farmer) is to distribute the tasks among the workers and gather the results.

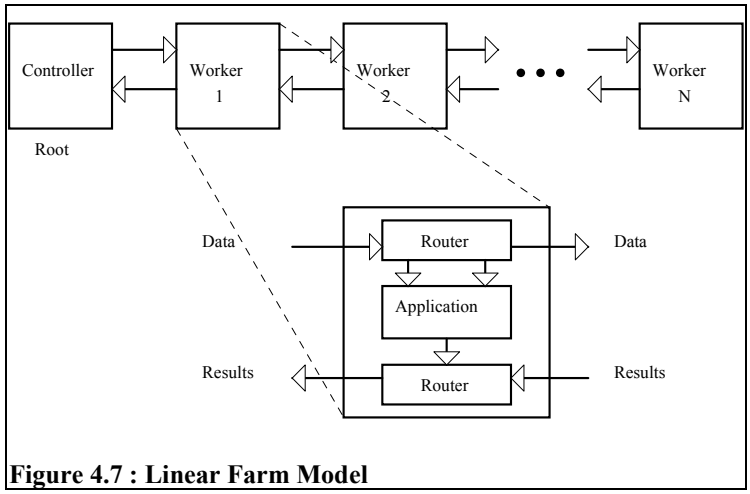


Figure 4.7 : Linear Farm Model

The application code (the mssm algorithm) is run on each of the worker processors. A simple harness is used to perform message routing through the network. The harness, which consists of just a few lines of code, consuming little of the processor time, is required to act as an interface between the application running on a particular processor and the rest of the network. In our implementation we have used the Cstools interface. This latter is a message passing system which establish communication between processes in many schemes (i.e. blocked, unblocked, and buffered). Furthermore, it allow us to consider the computing network (i.e.: Linear farm) as an abstract machine (a fully connected network) in which the communication between processes is independent on the hardware links between transputers.

4.2.2: Data partitioning and task granularity

The design of this parallel solution is based on partitioning the data (organ slices) into a 2D grid of segments that we call grains (g_t) as shown in figure 4.8. Then we consider the matching of these segments as tasks which are scheduled according to the process farm model.

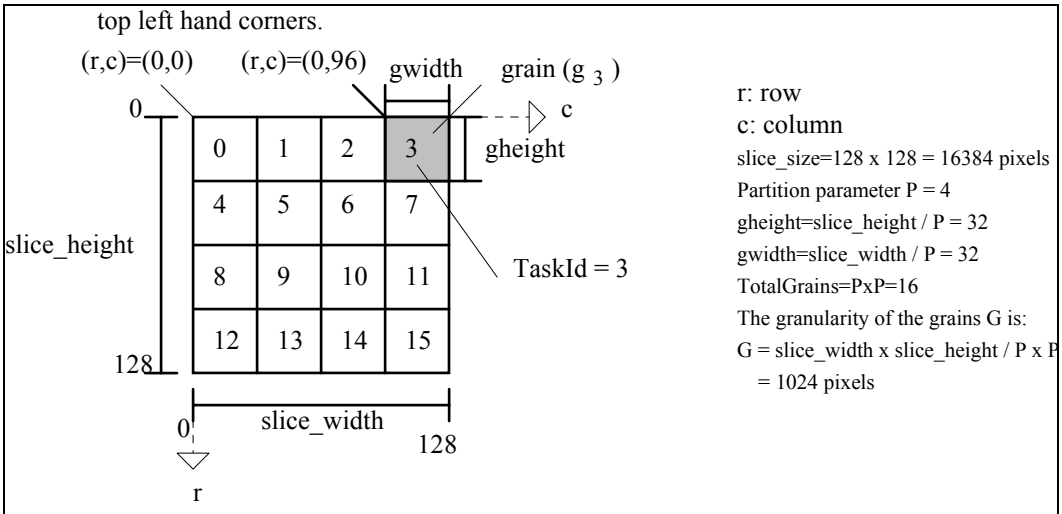


figure 4.8: Data partitioning and task granularity.

Since the slices are represented by two dimensional matrices (slice_*width x slice_height*) of pixels, the granularity (*G*) of each grain (g_t) is determined by the partition parameter (*P*). This parameter partitions the *slice_width* and *slice_height* into equal partitions called grain width (*gwidth*) and grain height (*gheight*) as follow:

gwidth = slice_width / P and grain height is gheight = slice_height / P Equ 4.1

The total number of grains involved in the matching process is:

$$\text{TotalGrains} = P^2 \quad \text{Equ 4.2}$$

Hence, the granularity of a grain (*G*) is equal to :

$$G = \text{slice\_size} / \text{TotalGrains} = (\text{gwidth} \times \text{gheight}) = (\text{slice\_width} \times \text{slice\_height}) / P^2 \quad \text{Equ 4.3}$$

The matching of a grain by a given worker process is called a *Task*. We refer to these tasks by the task identification numbers (*TaskId*) which are gathered in a table called *TaskQueue*. This table is build by the farmer based on the dimensions of the slices and the granularity of the grains. From the *TaskId* we can compute the top left hand coordinates referring to the rows and the columns (r,c) within the 2D image. This computation by the worker process based on the *TaskId* received from the farmer according to following equations:

r = TaskId/P) * gheight and c = mod(TaskId/P) * gwidth Equ. 4.4

As an illustrative example figure 4.8 shows the data partitioning of a 128 x 128 slice with a partition parameter P=4 which results in total number of grains equal to 16 grains and a granularity G=1024 pixels. The shaded window represents the grain (g_3) referenced by the *TaskId* number 3 with (r,c) = (0,96).

4.2.3 Matching windows and data requirements

Let us know consider the data required by a task in this parallel solution by referring to figure 4.9. A task matches two grains and outputs a discrepancy map (*gxymap*). This is to say that the mssm algorithm computes the *gxymap* for an individual grain in *N* matching iterations within the matching window (*MW*) which is defined by the grain size and the elasticity parameter *E*. The last matching iteration may find that the best match of an organ pixel with the model pixel is located at the edge of the matching window. Furthermore, even if we select the granularity of the grain as one pixel (*G=1*) , we have to filter all the pixels within the filtering window (*FW*).

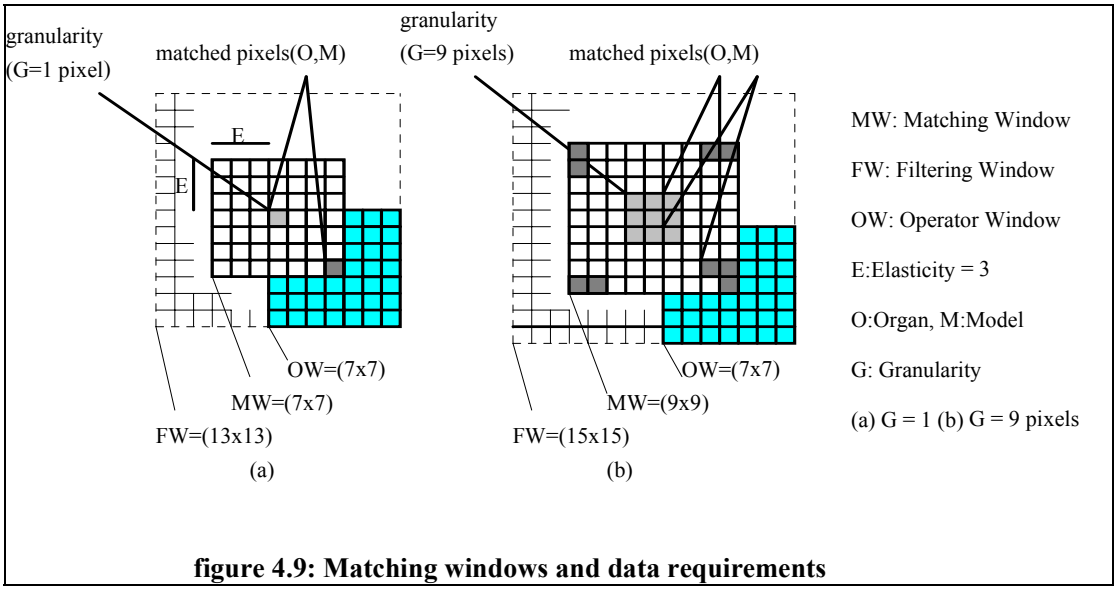


figure 4.9: Matching windows and data requirements

4.2.4: Data replication and locality of data access.

The farmer process sets all the parameters of the algorithm by reading the parameter file (input.dta) from the filing system. These parameters consist of the data files of the NMR images (organ and model slices), the Gaussian operators, the partition parameter P , the elasticity parameter E, and the number of vision channels V. Then it reads the corresponding data files and sends a copy of these images for *replication* in the local memory of each

worker processor as well as all the other parameters. The worker processes contribute in the data replication step by computing then saving the set of operators required by the (V) visual channels of the algorithm in the local memory of each worker processor.

Figure 4.10 shows how the data replications of the images and operators are duplicated in the local memory of each processor in the network. In this way, the farmer process do not send to the worker the data of the grain (g_t) but sends a structured message *fmessage* which comprises two fields a signal *fmessage.signal* and the task identification number of the grain *fmessage.taskid*. Once a worker process receives a message, it computes the coordinates (r,c) of the grain g_t according to equation 4.4, identifies the coordinates of the grain (g_t) in the local memory of the processor, performs its matching with the corresponding grain of the model, and finally sends the discrepancy map (*gxymap*) of the matched grain to the farmer process in a formatted message that we explain latter.

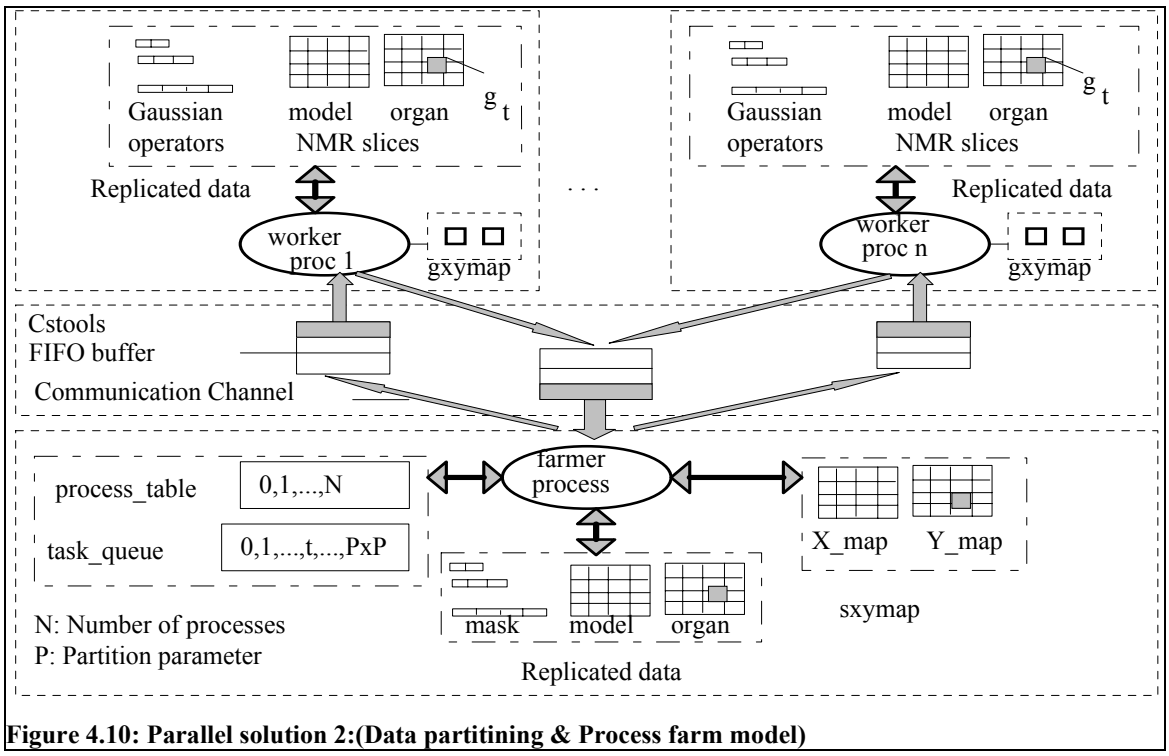


Figure 4.10: Parallel solution 2:(Data partitining & Process farm model)

This data replication has two advantages. The first, it eliminates the communication delay involved in the data access by worker processes by exploiting the locality of data access and reduces the length of the message packets. The second it allows the worker processes to compute without worrying about data integrity and avoids hot spots. However, the amount of memory required to replicate all the necessary data is a serious disadvantage. This last point may limit the scalability of this solution to larger data sets.

4.2.5: Task Scheduling (demand driven)

The farmer process builds the *TaskQueue* and registers all the worker processes in a table called *ProcessTable*. It then starts scheduling tasks to the ready worker processes in a demand driven fashion. At start the farmer process sends one or two *TaskIds* to all workers and waits for the first worker ready with a discrepancy map (*gxymap*). Following the reception of this later from a worker process, the farmer stores it in the slice map sxymap (This latter operation is considered as an implicit request from the worker for another *Task* if there still more tasks for processing) and sends the worker the next *taskId* from the *TaskQueue*.

The advantages of this scheduling technique is that it provides an automatic mechanism for dynamic load balancing among processors even for a heterogeneous network. In another word fast processors will not be penalized in the presence of slow processors, close processors in the network will wait for far away processors due to communication delays.

4.2.6 Synchronization using Buffered Messages

The synchronization between the farmer and worker processes is based on the Cstools message passing system (MPS). The transmitted messages are buffered at the receiver in a FIFO buffer as shown in figure 4.11. Each message is formatted with a number of fields which convey *synchronization flags*, *TaskIds*, and *processed data* if any.

The worker process receives from the farmer a structured message with the following fields: *signal* and *TaskId* as follow :

define struct fmessage { int signal; int taskid; } FM;

Then, it performs the matching of the *grain* referrenced by *TaskId*. Finally, it sends to the farmer process a message formatted with the following fields: process number (*procno*), (*TaskId*), and the grain discrepancy map (*gxymap*) as follow:

define struct wmessage {int procno, int taskid, char *gxymap} WM;

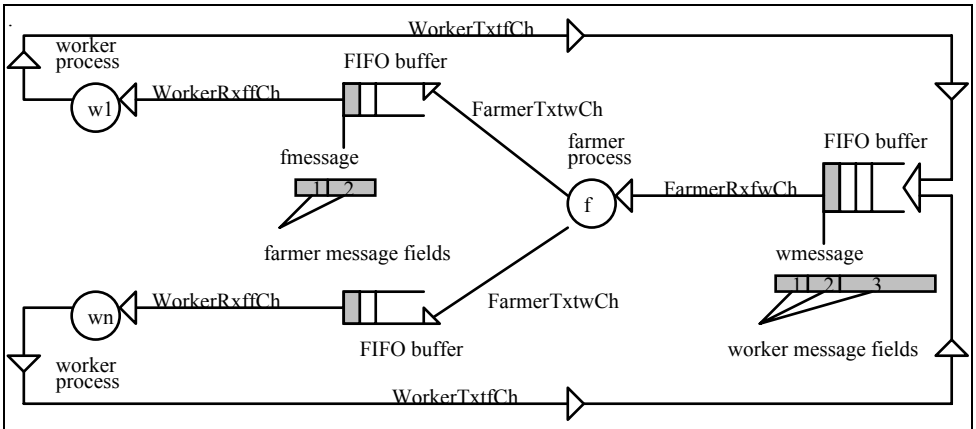


figure 4.11: synchronization with buffered messages.

The farmer process tries to keep the worker processes always busy by filling the buffer with two formatted messages. Once the worker finishes processing its actual task, it sends the result to the farmer and reads the next *TaskId* from the receiver buffer, in the meanwhile, the farmer is filling the buffer with the next *TaskId*. This buffering technique is provided by Cstools, where a sender sends its message through a named channel provided with a FIFO buffer (refer to figure 4.10 and program5).

4.2.7 Programming parallel solution 2 using Cstools

The programs of the parallel solution 2 using Cstools are described in program5(a) and 5(b). Program5(a) describe the implementation of the farmer process *farmer_process()* and program5(b) describes the worker process *worker_process()*.

```
farmer_process(modelPtr, organPtr,  Xdiscrep,Ydiscrep,GaussOperators,
                                                        V,G,E,width,height){
 FM *ftwmess;WM *wtfmess;                    /*define the structure  of messages*/
 Transport FarmerRxfwCh, FarmerTxtwCh; netid_t WorkerId;
 cs_getinfo(&nProcs,&ProcNo,&localId);
 cs_open(CSN_NULL_ID,&FarmerTxtwCh);
 cs_registername(FarmerRxfwCh,'farmer');
 initialize();                        /* read  parameters and initiatlize ProcessTable*/
                                      /* TotalGrains, TaskQueue, TaskPtr*/
 replicate(ProcessTable);             /* replicate data on the network*/
 initsend();                          /*send the initial  taskid to the processes*/
          /*.....Services the workers on request.....*/
 for(i = 1; i < TotalGrains; i++){
  csn_rx(FarmerRxfwCh,NULL, wtfmess,sizeof(WM));
  store(wtfmess);                          /*store gxymap int syxmap*/
  csn_lookupname(&WorkerId,wtfmess.procno);
  if (taskPtr <=TotalGrains) {             /*send next TaskId in the task_queue if still any*/
       ftwmess.signal=1;  ftwmess.taskid=*tasPtr;
  } else ftwmess.signal=0;
  cns_tx(FarmerTxtwCh,0,WorkerId,ftwmess, sizeof(FM));
  taskPtr + = 1;
 }
 save();                                   /*save the result in a file*/
}
```

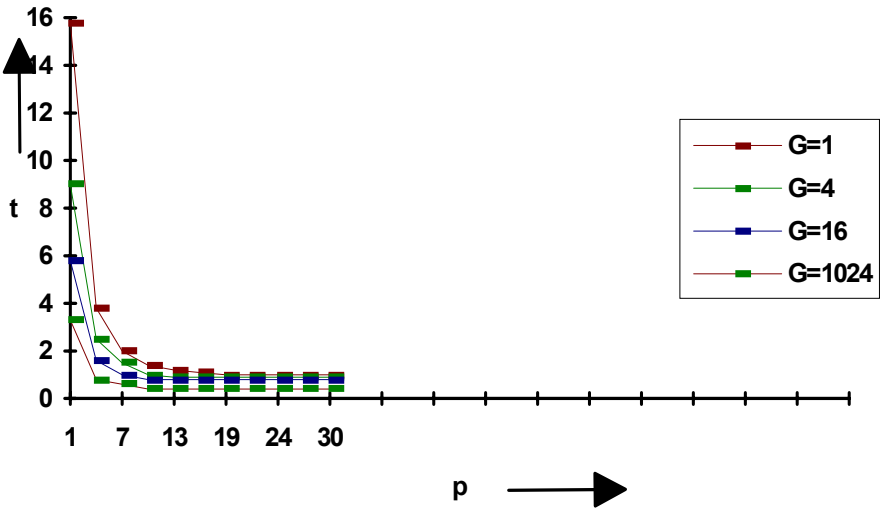
Program5(a):farmer_process.

```
worker_process(modelPtr, organPtr,Xdiscrep,Ydiscrep,GuassOperators,
                                                    V,G,E,width,height){
 FM *ftwmess;WM *wtfmess;                  /*define the structure of messages*/
 Transport WorkerRxffCh;
 Transport WorkerTxtfCh; netid_t FarmerId;
 cs_getinfo(&nProcs,&ProcNo,&localId);
 sprintf(wname,'worker%1d\0',procNo);
 cs_open(CSN_NULL_ID,&WorkerRxffCh);
 csn_registername,WorkerRxffCh,wname);
 cs_open(CSN_NULL_ID,&WorkerTxtfCh);
 csn_lookupname(&FarmerId,'farmer',1);
        /*worker register itself in process table*/
 wtfmess.procno=ProcNo;
 csn_rx(WorkerRxffCh,Null,ftwmess, sizeof(FM));
        /*worker enters the receive mode*/
 while(ftwmess.signal == 1){
   csn_rx(WorkerRxffCh,Null,ftwmess, sizeof(FM));
   initialize(gxymap);
   msm(organPtr,modelPtr,gxymap,ftwmess,V);          /* match ftwmess.taskid   */
   csn_tx(WorkerTxtfCh,0,FarmerId,wtfmess,sizeof(WM)); /* send gxymap to farmer*/
  }
  stop();
}
```

Program5(b): worker_process.

4.2.8 Performance of the parallel solution 2

The performance of this parallel solution is a function of the granularity *G* of the grains and the number of worker processes N. An optimum grain size is found to give the best performance as shown in figure 4.13 bellow. This is due to the matching process which requires extra filtering for fine grains, and to the task scheduler which fails to balance the load between the worker processes for large grains. We expect a saturation of the model after a number of processes equal to the total number of grains TotalGrains=Nprocs. Therefore, the system will reach its best performance for an optimum grain size and for a number of processors equal to the number of grains.



G=1,4,16,1024 pixels, V=8 visual channels, E=2 pixels.

Figure 4.13: Performance of the farmer model

The performance of this solution can be appreciated if we compare it to the parallel solution1. For this purpose we measure the following:

Pt=30 seconds , St=4 minutes, slice_size=16384 pixels, E=2 pixels, and V=8 visual channels.

The speedup obtained by this solution can be computed according to Equation 3.x as:

S=T(1)/T(N) =.125 which demonstrates the superiority of this parallel solution compared to parallel solution 1 which attained a maximum speedup of .33.

The granularity measure R/C (refer to section 3.2.2.2) should be large for high speedups, and it is demonstrated from figure 4.13 that for G=1024 we obtain the best speedups because the communication overhead is very small compared to the run time of processes. Indeed, for fine grains G=1 The R/C is very small and the speedup is at minimum.

The advantages:

The advantages and disadvantages of this parallel solution are listed bellow:

•The data replication eliminates the communication delay involved in the data access of the worker processes by exploiting the locality of data access and reduces the length of the

message packets and also allows the worker processes to compute without checking for data integrity and avoids hot spots.

- The scheduling technique provides an automatic mechanism for dynamic load balancing among processors even for a heterogeneous network. The farmer process tries to keep the worker processes always busy by filling the buffer with two formatted messages. This buffering lessens the worker processors idle time.

The disadvantages:

- The amount of memory required to replicate all the necessary data is a serious disadvantage. This last point may limit the scalability of this solution to larger data sets (i.e. 3D organs).
- For high number of worker processes (and fine grain), the controller itself becomes a bottleneck.

Improving the parallel design

The performances obtained by the concepts of data partitioning and farming is limited and the algorithm do not scale to volumetric data sets (refer to the requirements of the mssm algorithm chapter 1). To improve our design in performance and scalability we may consider the following. For high number of worker processes (and fine grain), it may be necessary to use a hierarchy of controllers to avoid the controller itself becoming a bottleneck. This bottleneck can be lessened if we further extend the farm model by configuring the system as a tree topology [Green88]. This extension permits to the farm model to allow work tasks to be completed when data items not resident locally at a node can be obtained by making a request for data across the network. This will allow us to save memory space and allow the algorithm to scale up with the parallel machine resources.

4.3 Conclusion

In this section we discusses the performance of the designed parallel solution and shows the limitations of the parallel models used in the design. Thereafter, the author argues

that in order to design an efficient parallel solution over MIMD networks we have to promote the network to support shared memory. These concepts will be grasped from existing MIMD Shared Memory Parallel Computers [Gottlieb83] which are based on message switching networks, higher concepts for process synchronization [Valiant89], and the support of shared memory abstraction[Bisiani88].

Finally the author, argues that a parallel solution can be obtained for the elastic matching algorithm over the transputer network with the ANSIC toolset parallel compiler[Inmos90a]. This compiler supports both Communication Sequential Processes (CSP) and Shared Memory (SM) primitives. Therefore, it is possible to provide shared memory access based on a memory coherency policy [Nash91] and allow the algorithm to scale up.

The key to an efficient implementation of the multiple scale elastic matching algorithm is a scaleable shared memory. Scaleable shared memory allows the implementation of mechanisms to achieve both performance and scalability of application algorithms on a general class of MIMD machines. It allows the exploitation of parallelism inherent in the algorithm and perform process synchronization by shared memory operations. It also allows the storage and manipulation of large volumetric data sets in shared memory. These mechanisms, combined with scaleable data structures, allow the design of an efficient scaleable parallel version of the algorithm.

MIMD Shared Memory Parallel Computer which provide the mechanisms of process synchronization by shared memory do exist such as the NYU Ultracomputer [Gottlieb83], several other DSM machines such as Alewife, Dash and Stanford Dash are surveyed in [Krishnamoorthy94]. When programming applications with shared object for instance over the transputer networks [May90], shared objects offer an abstraction for shared memory in which programming flexibility is traded in favour of performance and scalability. Other

programming models based on BSP provide dynamic process creation , fetch-and-op,s operations and memory coherence [Valiant89].

The author is actually experimenting the bulk synchronization technique and parallel slackness under the ANSIC toolset [Inmos90b] over a linear transputer network [Green88] hosted on an IBM PC.

Chapter 5

Commentaire [b1]:

Mating LINDA, BSP and SM parallel programming models with the MSSM algorithm over distributed memory MIMD networks

Contents

5.1 Introduction
5.2 Designing with Linda parallel programming model (Uncoupled programming)
5.2.1 The model
5.2.2 Uncoupled programming (The interface)
5.2.3 Programming with C-Linda the elastic matching problem
5.2.4 Performance of the model
5.3 Shared objects (abstract data types)
5.3.1 The model
5.3.1.1 Shared Objects
5.3.2 The interface
5.3.2.1 Data structures
5.3.3 Programming the elastic matching problem with SO model
5.3.4 Performance of the model
5.4 XPRAM programming model
5.4.1 The Underlying Machine Model
5.4.2 The XPRAM programming model
5.4.2.1 Process management
5.4.2.2 Memory access
5.4.2.3 Shared space
5.4.2.4 Process synchronisation
5.4.3 A programming interface for the XPRAM model
5.4.3.1 Process management
5.4.3.2 Shared memory
5.4.3.3 Process synchronisation
5.4.3.3.1 Futures
5.4.3.3.2 Bulk synchronous operation
5.4.4 Programming with XPRAM model
5.4.5 Performance of the model

5.1 Introduction

From chapter 4, we have concluded that the design of an efficient solution to the mssm algorithm over distributed memory systems requires advanced parallel programming models [Harris94, Kung88, Valiant90, Williams90, Zair92]. This is necessary to achieve an effective balance of computation, memory and communication appropriate to the requirements of the algorithm over a wide range of multicomputers MIMD target machines [Hockney88, Stone87].

The author argues that efficient solutions could be reached by promoting the MIMD machine to support shared memory access [Bisiani88, Krishnamoorthy94]. Shared memory access can be provided in a distributed memory system (DSM) by using techniques such as hashing [Jones92] to provide coherent shared memory [Bisiani88]. Furthermore we can have a fully connected network supporting shared memory (clarify the underlying model) . Therefore we can use parallel programming which allow us to use advanced concepts such as Barriers [Valiant89], Shared Objects [Mallon91], Generative Communication [Gelernter85], Shared memory [Berhad91], Concurrent Objects [Zair92], etc....

However, we present in this chapter three new solutions to the mssm algorithm with selected representatives of the previous parallel programming models:

Associative memory (LINDA [Ahuja88, Carriero88]),

Shared Objets (SO [Mallon91]), and

Bulk Synchronous Parallel (XPRAM [Valiant89, Nash91]).

The three solutions are designed with the model's interface of C-Linda [Ahuja88], SO [Mallon91] and XPRAM [Nash91]. Each solution is presented with a graphical diagram followed by C programs using the model's interface.

From this work we have experienced the advantages of each of them in promoting the parallel machine to support the requirements of the algorithm. Those requirement that have been identified as the following: (1) the complexity of the multiple scale signal matching (**MSSM**) algorithm (Pipeline nature), (2) the large data sets of the medical objects used [Flynn83, Goldwasser87, Kennedy87), and (3) the constraints of the data interdependencies (filtering windows).

5.2 Designing with Linda parallel programming model (Uncoupled programming)

5.2.1 The model

Linda model is based on generative communication [Ahuja88, Carriero89, Gelernter85]. If two processes need to communicate, they don't exchange messages or share a variable; instead, the data producing process generates a new data object (called a tuple) and sets it adrift in a region called tuple space (refer to figure 5.1). The receiver process may now access the tuple. Creating new processes is handled in the same way: a process that needs to create a second, concurrently executing process generates a "live tuple" and sets it adrift in tuple space. The live tuple carries out some specified computation on its own, independent of the process that generated it, and then turns into an ordinary, data object tuple.

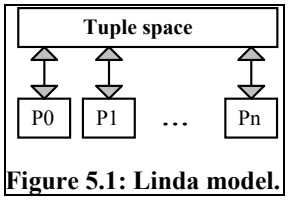


Figure 5.1: Linda model.

Linda deals only with process creation and coordination. If a modular language is embedded in the Linda model, Linda becomes part of a "modular" approach to parallelism; likewise with an object oriented language, or anything else. By not meddling in computation

issues, Linda wins the freedom to coexist peacefully with any number of base languages and computing models, and to support clean, simple, and powerful operations within its own domain.

5.2.2 Uncoupled programming (The interface)

Linda provides four basic operations, **eval** and **out** to create new data objects, **in** and **rd** to remove and to read them respectively. If a Linda sending process S has data for a receiver R, it uses out to generate a new tuple. Then R uses **in** to remove the tuple. A tuple, unlike a message, is a data object in its own right. In message sending systems, a message must be directed to some receiver explicitly, and only that receiver can read it (unless the programmer uses some special broadcast message operation, which some message systems supply and some don't). Using Linda any number of processes can read a message (i.e. a tuple); the sender needn't know or care how many processes or which ones will read the message.

The fact that senders in Linda needn't know any-thing about receivers and vice versa is central to the language. It promotes what we call an uncoupled programming style. When a Linda process generates a new result that other processes will need, it simply dumps the new data in tuple space. A Linda process that needs data looks for it in tuple space. In message passing systems, on the other hand, a process can't disseminate new results without knowing precisely where to send them. While designing the data generating process, the programmer must think simultaneously about the data consuming process or processes.

5.2.3 Programming with C-Linda the elastic matching problem

To program the elastic matching problem using linda, we will use C-Linda [Carriero89]. Process p0 (*specify that p0 is the root processor*) divides the images into segments (refer to chapter4) and dumps them (using **out**) in the tuple space as segment_tuples *(It seems that segment tuples do not satisfy the data requirements for the filtering windows?)*, builds the masks

and dumps them as mask_tuples, and builds the task_queue and dumps it as task_tuples (refer to figure 5.2). All processes from p0 to pn remove a task from the task_tuples (using **in**), reads the segment_tuple from the segment_tuples, compute the segment discrepancy map (dis_segment) and dumps it into the discrepancy map tuples dis_segment_tuples. Processes have simultaneous access to the tuples and communication is not performed between processes but through the TS operations **in**, **out**, and **rd**.

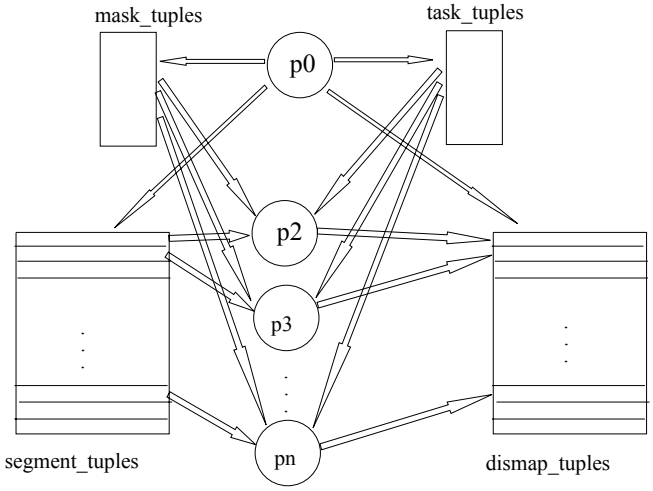


Figure 5.2: Programming the elastic matching problem using Linda.

```
Tuple_process(){

 while(get(new_segment)){
   out("task",new_segment);
   in("task",?coordinate,segment);
   if(++task > Max_number_of_tasks)
   match model with image→
"dis_segment");
     do{  out( in("result", ?result);
            update the discrepancy map;
          } while (--tasks >0)
  }
  while(tasks--){ in("result",?result);
                  update the discrepancy
map;}
generate new_image;
}
```

Program 5.1 (a):The design in C-Linda.

msm_process(){

```
do{
   in("task",?coordinate,segment);
   match model with image→"dis_segment");
   out("discrepancy_map",coordinates,
                                dis_segment);
  }  while(match is not complete);
}
```

Program 5.1 (b):The design in C-Linda.

5.2.4 Performance of the model

There is no bottleneck at any stage of the network. There is a great saving in memory since the data structure is not replicated. Therefore, the application can scale up to large data sets. The speed up of execution is only limitted to the interface response time.

5.3 Shared objets (abstract data types)

5.3.1 The model

The SO model [Mallon91] falls somewhere between the traditional extremes of message passing and shared variables, and is based on shared abstract data types. Its purpose is to provide a shared variable abstraction in distributed memory machines; which maintains the performance and scalability characteristics of message passing, rather than those usually associated with virtual shared memory.

Shared Objects (SO) is different from shared memory abstraction because it doesn't provide support for general shared data structures. Instead, specific abstract data types-for example; trees, queues, grids, lists and bags- are provided which, although physically distribute around the system, appear as autonomous, concurrently accessible objects.

Shared objects is different from other hybrid systems; for example, Linda[Carriero89], Emerald[], Orca[] and CMU's Concurrent Objects[Gehringer87], because it is machine, rather

than application, oriented. It is concerned with providing an efficient low level abstraction for shared data management. Most hybrid systems are aimed at providing high level programming abstractions which simplify parallel programming (hiding complex details, or becoming application or methodology specific). For example, Linda provides the 'generative communication model' of tuple space to decouple processes in time and space.

5.3.1.1 Shared Objects

In Shared Objects a parallel program is a collection of sequential threads of execution called processes which communicate through shared abstract data types called objects. Programs are written in a conventional sequential language such as C or fortran, and linked to a pre-defined library of shared abstract data types. The types are instantiated and accessed via explicit call interface.

A contiguous shared address space spans all processes in the system; at each location an instance of one SO's pre-defined types may be created. This contrasts with the associative memory of Linda which addresses data via structured naming, and with more conventional models of shared memory which provide both a contiguous address space and a contiguous memory space.

Shared objects are instantiated and destroyed by calls to *Create* and *Destroy* respectively. When creating an object the programmer may specify the shared address for, and access patterns of, the instance. For example, placing a new instance at shared address 102, and indicating that it is read more often than written. The shared address is a unique *handle* for the instance, and must be used whenever a process wishes to interact with it. The memory looks the same to all processes; any process can create an instance, and once it exists, any process can access and modify it.

5.3.2 The interface

Interaction with an object occurs through a call interface consistent with the abstract data. There are nine abstract data types supported by SO, and nine corresponding logical names to identify them. The logical names are defined in the standard header <so.h>: QUEUE, PRIQUEUE, STACK, ITEM, GRID, TREE, BAG, BALBAG, and LIST.

The **Queue** data type provides operations to *enqueue* and *dequeue* elements, the **Stack** data type to *push* and *pop* elements. Operations on the abstract data type Bag are performed by the functions: **Put()**; **Get()**; **BGet()**; **EGet()**; **Randomise()**.

The interface specification for the BAG is as follow:

int Put(int address,void *data,int length).

Grid objects are unlike most other SO data structures in that their bounds are fixed. When instantiated, all elements of the Grid are unset ; the contents of each location is not simply undefined, but is without a value. If a process attempts to read an unset location, then an error is returned; however, if a blocking read is attempted, then the process is suspended until the location is set- assigned a value. This property of synchronising through Grid elements is particularly useful for blocking read and "Multireel" operations. Here are the allowed functions: Write(); Read(); Remove(); Multireel(); Bread(); BRemove(); Swap_Row; Swap_Col; Set(); Unset().

The interface specification for the GRID Write function is as follow: int Write(address, row,row_cnt,col,col_cnt,data).

5.3.2.1 Data structures

Shared Objects provides data structures to support a variety of message passing abstractions and to support shared variables. Objects are not instantiated on particular nodes, but

are physically distributed around the system, and logically accessible by shared addresses. Mechanisms support concurrent access and synchronisation -usually in keeping with the abstract data - and techniques are employed to eliminate bottlenecks (central controllers) and minimise communication overheads by reducing all such operations (i.e. Combine several dequeues into one). Some object types are sophisticated; for example, Balancing Bags Back-end many load balancing algorithms by handling task distribution, and Grids provide multireel operations on rows, columns or any sub-grid. In effect, SO provides a unifying structure in which message passing, shared variable and load balancing are integrated within a single model [Nash91].

5.3.3 Programming the elastic matching problem with SO model:

This solution is based on the use of distributed data structure through shared abstract data types. The shared data structure enables us to establish interprocess communication, reduce communication overhead and eliminate the bottleneck over the central controlle (refer to figure 5.3). The patient and model images are written into the corresponding Grid_Patient and Grid_Model , for which all processes can accesses concurrently. The X and Y discrepancy maps are also Grids Grid_X and Grid_Y; in which the processes write their result.

The Masks are computed by the processes and written into the Grids: Grid[0],Grid[1], ..., Grid[Num_of_filters].

Replicated processes compute the tasks to be performed for the whole matching process and writes them into a BAG; The Bag gives no preferable order in executing the tasks (refer to program 5.2(a) and 5.2(b)).

The worker processes do not have to receive from any controller which task they have to compute, instead they get a task from the Bag_Task. The process reads the Grid_Patient,

Grid_Model, and the Grid_Mask[i] and write the result in Grid_X and Grid_Y; and gets another task from the Bag_Task if there still any.

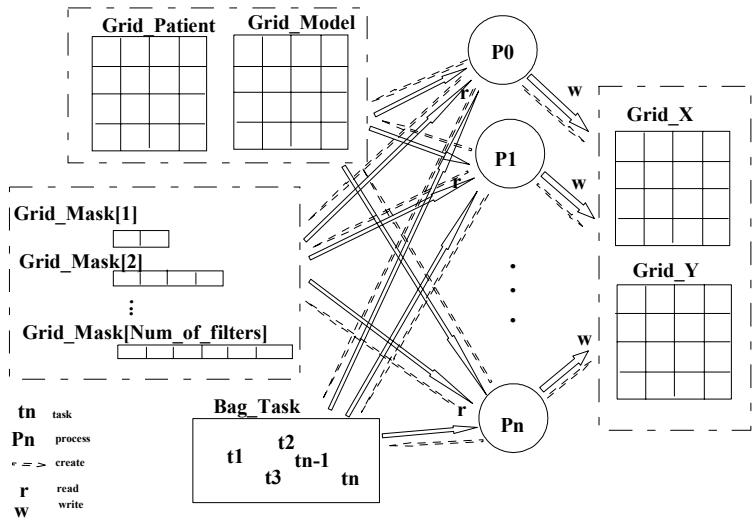


Figure 7:Shared Objects model of computation for elastic matching.

instantiate(){ /**This function instantiates the shared abstract data types, and fills them with data. replicated processes cooprerate in the instantiation and putting data instances.**/

/**instantiate Grid_P,and fill it with patient_image: p_im; Grid_M and fill it with model_image m_im; Gird_M , Grid_X, and Grid_Y */

```
Create(Grid_P,GRID,R, width, length);
read_image(p_im,&ras,&imPtr,8,NULL);
Write(Grid_P,0,width-1,0,length-1, &imPtr
);
Create(Grid_M,GRID,R, width,length);
read_image(m_im,&ras,&mdPtr,8,NULL);
Write(Grid_M,0,width-1,0,length-1,
&mdPtr)
Create(Grid_X,GRID,W,width, length);
Create(Grid_Y,GRID,W,width, length);

/*Instantiate the Grid_Mask[Nb_filters] */
for(i=0,i<Num_of_filters,i++){
Create(Grid_Mask[i],GRID,R,1,msize[i]);
Write(Grid_Mask[i],0,msize[i],0,0,mPtr[i]);
}

/* Instantiate the Bag_Task */

Create(Bag_Task,BAG,R,Nb_segts,UNDEF
);

for(i=0;i<Num_of_Column_segments;i++){
   for(j=0;j<Num_of_Row_segments;j++){
      task[0]=i; task[1]=j;
      Put(BagTask, &Task, sizeof(task));
   }
/*end of instantiate*/
```

} **Program 5.2(a): instantiate function.**

```
replicated_process()
/* instantiate and build the shared abstract
data types */

instantiate(Grid_Patient,Grid_Model,
   Grid_X,Grid_Y,&Grid_Mask,Bag_task,
   &mask_size,Num_of_segments,width,
   length,patient_image,model_image,
   imagePtr,modelPtr,&MaskPtr,
   Num_of_filters,segment_size)

/* address wait the shared objects
instantiate by the replicated processes*/

for(addr=0;addr<Nb_addresses;addr++{

retCode=AddressWait(addr,ALLOCATED);
  if(retCode !=OK)
    exception(""AddressWait",retCode);
 }
```

```
 /*Get a task from the Bag and process it*/

 while(retCode !=0) /*Get returns 0 if the
Bag is empty*/
 {
  retCode=Get(Bag_Task,task,sizeof(task));
  /* compute the discrepancy map*/
for(count=Nb_stages-1; count>=0;count--){
   filter(count,task,Grid_Patient);
   filter(count,task,Grid_Model);
   match(count,task,Grid_Patient);
  }
/* Put the discrepancy map into Grid_X and
Grid_Y*/
Write(Grid_X, &task, segmentsize,
&task+1, segmentsize, &Xpointer);
Write(Grid_Y, &task, segmentsize,
&task+1, segmentsize, &Ypointer);
 }
```

Program 5.2 (b): Object-Oriented listing.

5.3.4 Performance of the model

the performance of this model can be stated in the following:

-Shared abstract data types have saved memory compared to the message passing.

-This solution has eliminated the need for a controller which resulted in a reduction in communication overhead.

5.4 XPRAM programming model

5.4.1 The Underlying Machine Model

The machine class efficiently supporting the XPRAM model is the class of distributed memory MIMD machines with an interconnection network able to combine read and certain update requests for the same data item, in a similar way to the NYU Ultracomputer's fetch-and-add primitive [Gottlieb83]. Figure 5.4 presents the physical machine of the XPRAM model. It

reflects a class of physical machines, primarily message passing and shared memory MIMD computers. The performance model for the machine is characterised by the following parameters:

-A process may execute arithmetic and logical operations, deschedule itself, reschedule a process on the same node or send a request to the I/O processor in O(1) time.

-The network has a mean distance D between two nodes (measured in network links), and a capacity C(the total number of links in the machine).

-The model includes a virtual shared address space implemented by the randomisation of the shared data (on a word or multiple-word basis) across the memory modules using a hash function (assumed to be a linear polynomial computable in O(1) time).

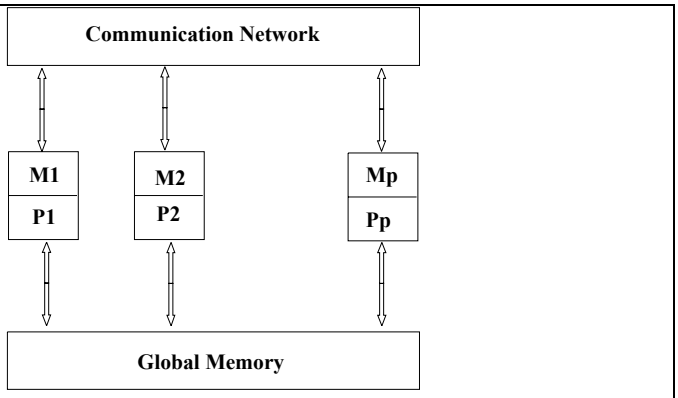


Figure 9: Physical machine model. It reflects a class of physical machines (Primarily message passing and shared memory MIMD computers).

5.4.2 The XPRAM programming model

The XPRAM is related to the PRAM [Harris94], [Gibbons89], but differs in a number of important respects. Firstly, the processors do not operate in lockstep. They operate asynchronously, but have the ability to explicitly synchronise as in Gibbons Asynchronous PRAM. Furthermore, data written to shared memory will not be guaranteed visible to other

processes until the originator of the data has carried out a synchronisation (or *coherency*) operation. Thus the shared memory is *weakly coherent*. This is similar to the use of messages to guarantee coherency described by Bisiani and Istavrinos[Bisiani88]. The cost of the shared data access has been shown in the previous section to be very similar to Chin's model. The ability of processors to compute asynchronously and a weakly coherent memory model allows for more efficient implementation and corresponds more closely to the physical machines that are being proposed by May[12,14] as scalable parallel computers.

5.4.2.1 Process management

Process management in the model is based on the idea that a group of identical processes may be executed in parallel. This is achieved in three ways:

-locating the processes nondeterministically across the machine.

-located on the same processor as the calling process and the rest of the process group.

-created so that one process is placed on each processor in the machine.

The first two methods provide a way for an algorithm to hide the communication latency encountered in any physical realisation of such a model, called *parallel slackness* by Valiant.

5.4.2.2 Memory access

The XPRAM assumes a CREW model of memory access augmented with a limited *concurrent update* facility using a small set of associative binary operations (such as addition), since they have a wide applicability [Gottlieb83]. These represent operations which may be combined by the network on their way to the same shared address and help to reduce the possibility of hot spots in the network, contention at memory modules, the need to lock data items for exclusive access, giving a reduction in the completion time of a group of such operations executed in parallel.

5.4.2.3 Shared space

The XPRAM does not include any notion of a process address space, only a single shared space which has a number of data visibility and coherency rules. Reading or writing from or to shared address involves the transfer of the required data between the shared address space and the memory module on the process's node. Each update to a shared data item is immediately visible to the updating process, but has a number of possible visibility rules for other processes, depending on the manner in which it was created. A piece of shared data is defined upon creation as being either:

-Global: updating this data means that it will be visible to all processors after a coherency operation.

-local: updating or deallocating this data is only allowed on the processor on which the data was created and is only guaranteed visible to processes on that processor after a coherency operation. This is used in conjunction with process groups created on the same processor.

-private:updating or deallocating this data is only allowed by the process which created the data. Updates are only guaranteed visible to this process and so are not effected by any coherency operations. This effectively provides a process address space whilst keeping the simplicity of a single shared space.

5.4.2.4 Process synchronisation

The model provides three methods for process synchronisation:

-Through the use of the primitives associated with process management, consisting of the blocking of a process until either a certain process group or any process group created by that process has terminated.

-Bulk(or barrier) **synchronisation** which provides a guaranteed point at which no process within a process group may pass until all other processes within that group have arrived.

-Futures, which may be in a *set or unset* state and are shared between a pair of processes. A process blocking on an unset future will only unblock when another process has changed the futur's state to being set. This is achieved either explicitly, or giving the future a shared address value.

5.4.3 A programming interface for the XPRAM model

5.4.3.1 Process management

The process management is performed by the following operations, fork(), setup(), join(), first(), my_index(), nodes(). Fork() creates identical processes and places them according to the control fork_type, setup() creates a single identical process on each processor. Join() synchronise with the termination of a group, first() waits on the first group to terminate, my_index() returns a unique integer of the number of members in the group minus one, nodes() derives the number of processors in the system.

5.4.3.2 Shared memory

The static and dynamic allocation of shared data is performed by the following operations, global<type> g, local<type> l, private <type> p, char *gmalloc(int ng), char *lmalloc(int nl), char *pmalloc(int np). Free(char*address) releases a shared data area.

The concurrent updating of a shared item is performed by the following operations, read&add(), read&swap_addr(), read&swap_int(), read&or(), read&and(), read&max().

5.4.3.3 Process synchronisation

5.4.3.3.1 Futures

A future is declared using future f, created using create_future(), and freed using free_future().

The types handled by create_future() operation may be either GLOBAL_T (accessible across the machine), or LOCAL_T (within the same processor). Defining the visibility of the future at run_time allows data structures to be kept independent of global or local instantiation.

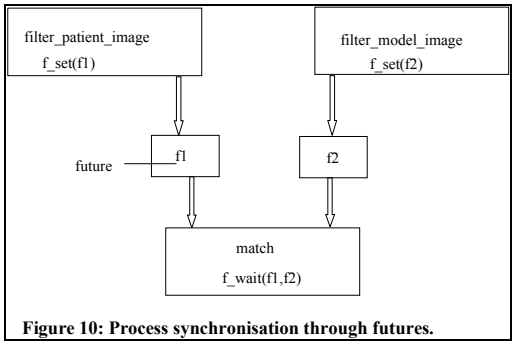


Figure 10: Process synchronisation through futures.

5.4.3.3.2 Bulk synchronous operation

Sync() is the synchronisation operation carried out by each process within a process group. When all processes have executed this then all may continue. This operation may be reused within a loop. The execution of sync() guarantees that the process group will be able to use the shared data produced within that group afterwards.

The synchronisation through futures is performed by the following operations: f_set(), f_unset(), f_wait(), f_bset(), f_write(), f_read(), f_bwrite().

5.4.4 Programming with XPRAM model

In our application data flows through a number of pipeline iterations, each one performing its task in two stages, filtering than matching of the patient and model images to produce a discrepancy map. The later is forwarded to the next iteration in the pipeline. The filtering stage of one iteration may be executed in parallel with the matching stage of the next iteration. Therefore,

we can exploit the pipeline nature of the algorithm and use the XPRAM model to fork a group of processes and synchronise between them using futures and join operation. The filter_image and filter_model processes sets correspondingly two futures f1 and f2 which will be inherited by the matching_process as shown in figure 5.5.

In this solution we will determine the bulk time by the number of filters. If Num_of_filters is equal to 3, then we will fork 6 processes which will not be relinguished until they have terminated processing all the stages of the pipeline for a given segment.

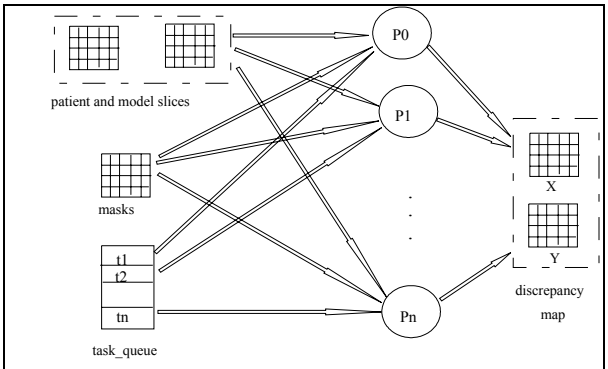


Figure 11: XPRAM model of computation.

The design of the solution is shown in program 5.3. The main program main() instantiate the data structure and creates a number of worker processes worker() equal to the number of segments using setup() on each node of the network. The worker process forks on its turn on the local processor of each node using fork(). The synchronization is established using the two futures f1,f2 created by the worker process when forking the filter and the match processes. We notice that the filter processes perform f_set(f1) or f_set(f2) and the match process performs f_wait(f1,f2).

```
main(){
....
Num_of_Processors=nodes();
global int image[width*length],

model[width*length];
global int X_D[width*length],

Y_D[width*length];
global int mask[Num_of_filters],
task_queue[Num_of_segments];
/*instantiate the data stucture*/
imagePtr=char *gmalloc(width*length);
retCode=read_image(patientname,&ras,

&imagePtr,8,NULL);
if retCode <0 {
     print_image_error(retCode,patientname);
     exit(1);
    }
....
build the task_queue;
Num_of_Processors=nodes();
/*build task_queue */
global int task_queue[Num_of_segments-1];
for(i=0,i<Num_of_segments;i++)
     task_queue[i]=segment_size*i;
setup("worker",G1,"VVVVVV",image,mod
el,

masks,X_D,Y_D,task_queue)
join(G1);
....
generate the new_image;
  }
```

```
worker(imagePtr,modelPtr,X_DPtr,Y_DPtr,
                                task_queuePtr){
....
future f1,f2;
f1=create_future(GLOBAL_T);
f2=create_future(GLOBAL_T);
global int temp1[segment_size],
           temp2[segment_size];
if (task_queue is empty)
        deschedule the worker process;
pick up a task from task_queue;
for(i=0;i<Num_of_filters;i++){
fork(filter(i,image),LOCAL,G11,1,"VVVF",

image,temp1,f1);
fork(filter(i,model),LOCAL,G11,1,"VVVF",

model,temp2,f2);
if (i>0) then join(G12);  /* filling the pipe*/
fork(match(i,dismap(i-1)),LOCAL,G12,1,

"VVVVFF",temp1,temp2,dismap(i),f1,f2);
}
...
}

filter(i,slice,maskPtr)
   {
    ....
    filter the slice with mask(i);
    if (slice=image)
      then f_sef(f1);
      else f_sef(f2);
   }

match(i,dismap)
   {
    ...
    f_wait(f1,f2);
    match the patient slice with the model;
   }
```

Program 5.3: Programming with the XPRAM.

The worker processes access the local memory for their processing and they access the global memory only to save the X_discrepancy and Y_discrepancy maps. The farmer process creates the task queue, creates the replicated processes and determine the bulk step.

5.4.5 Performance of the model

The XPRAM parallel programming model allowed to combine different style of computations as well as its ability to allocate newly created processes to processing elements dynamically, we have finally achieved an effective balance of computation, memory and communication.

The main program forks dynamically a worker process on each node of the system. Furthermore, the worker process forks on its turn a number of processes nondeterministically on the local processors on each node. This allows this solution to scale on a broad range of multucomputers MIMD machines.

Chapter 6
Parallelization of the Octree visualization algorithm

6.1 Introduction
6.2 The algorithm
    6.2.1 The octree visualization algorithm
    6.2.2 The application software Parallel Viewer Software (PVS)
6.3 The parallel model
6.4 Data granularity, and task scheduling
6.5 Messages and their protocols
6.6 Process synchronization of the process farm model
    6.6.1 Process spawing and channel links :CreatePlist(), CreateChannels()
        connection of the network with channels appch.c
    6.6.2 The farmer process
    6.6.3 The worker process
6.7 Design of the functions of the Parallel Viewer Software (PVS)
    6.7.1 Acquisition and AdjustView
        6.7.1.1 Acquisition
        6.7.1.2 Adjust View
    6.7.2 The octree visualization algorithm (Vision pipe)
        6.7.2.1 Encoding:BuildOctree(), InitOctNode(), TestOctant()
        6.7.2.2 Mapping: OctToQuad(), BuildQuadNode(),
            DestroyQuadKids(), InitQuadNode().
    6.7.3 Display : TraverseQuadtree(), FillBuffer()
6.8 Implementation
    6.8.1 The Parallel Viewer Software PVS version 3
    6.8.2 Globals and function prototypes pvs.h
    6.8.3 Graphics library
    6.8.4 Network library
    6.8.5 Data partitioning library: SavePgrain(), etc...
    6.8.6 Vision Library
6.9 Performance of the parallel solution
6.10 Conclusion

6.1 Introduction

This chapter presents the design of a parallel solution to the octree volume visualization algorithm [Cho93]. The octree visualization algorithm listed in chapter 2, program 2.2 uses the octree visualization technique to visualize a 3D volume of data through a vision pipe composed of three modules : encode, map, and display. This vision pipe is time consuming and justify the use of parallelization to obtain better speedups.

The octree encoding scheme [Samet88] partitions the volume data into eight sub-volumes. These sub-volumes are tested for homogeniety, and if they are heterogenous they are partitioned on their turn to eight other sub-volumes. This partitioning process is carried recursivelly until we reach a graphical resolution adequate to the medical application.

We have selected the process farm parallel model [May87] and data parallelism to parallelize this algorithm. We have designed a solution for an attached MIMD network of IMS B008 transputer boards [refer to section 3.5]. We have used the ANSIC parallel compiler to design the parallel solution and code it. The host machine was an IBM PC running DOS.

We have developed a software application called PVS with the octree display algorithm in its heart. This software is designed to provide an application based on the algorithm. The software data flow of this application is shown in figure 6.2. The main modules through which the processing of the data geos through are :Acquisition, AdjustView, and vision pipe (encoding, mapping, and display). The Visualization pipeline is in short : Octree encoding: *BuildOctree(),* Mapping octree to quadtree: *OctToQuad() and* Reconstruct the 2D frame buffer: Display().

6.2 The algorithm

6.2.1 The octree visualization algorithm

When an octree representation is used for the viewing volume, hidden-surface elimination is accomplished by projecting octree nodes onto the viewing surface in a front-to-back order. If you refer to figure 2.11 in chapter 2, the front face of a region of space (the side toward the viewer) is formed with octants 0, 1, 2, and 3. Surfaces in the front of the these octants are visible to the viewer. Any surfaces toward the rear of the front octants or in the back octants (4,5,6, and 7) may be hidden by the front surfaces.

Back surfaces are eliminated, for the viewing direction given in fig 2.11, by processing data elements in the octree nodes in the order 0, 1, 2, 3, 4, 5, 6, 7. This results in a depth-first traversal of the tree, so that nodes representing octants 0, 1, 2, and 3 for the entire region are visited before the nodes representing octants 4, 5, 6, and 7. Similarly, the nodes for the front four suboctants of octant 0 are visited before the nodes for the four back suboctants. The traversal of the octree continues in this order for each octant subdivision.

When a color value is encountered in an octree node, the pixel area in the frame buffer corresponding to this node is assigned that color value only if no values have previously been stored in this area. In this way, only the front colors are loaded into the buffer. Nothing is loaded if an area is void. Any node that is found to be completely obscured is eliminated from further processing, so that its subtree are not accessed.

Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according to the

view selected. We assume that the octree representation is always set up so that octants 0, 1, 2, and 3 of a region form the front face, as in figure 6.1.

A method for displaying an octree is first to map the octree onto a quadtree of visible areas by traversing octree nodes from front to back in a recursive procedure. Then the quadtree representation for the visible surfaces is loaded into the frame buffer. Figure 6.1 depicts the octants in a region of space and the corresponding quadrants on the view plane. Contributions to quadrant 0 come from octants 0 and 4. Color values in quadrant 1 are obtained from surfaces in octants 1 and 5, and values in each of the other two quadrants are generated from the pair of octants aligned with each of these quadrants.

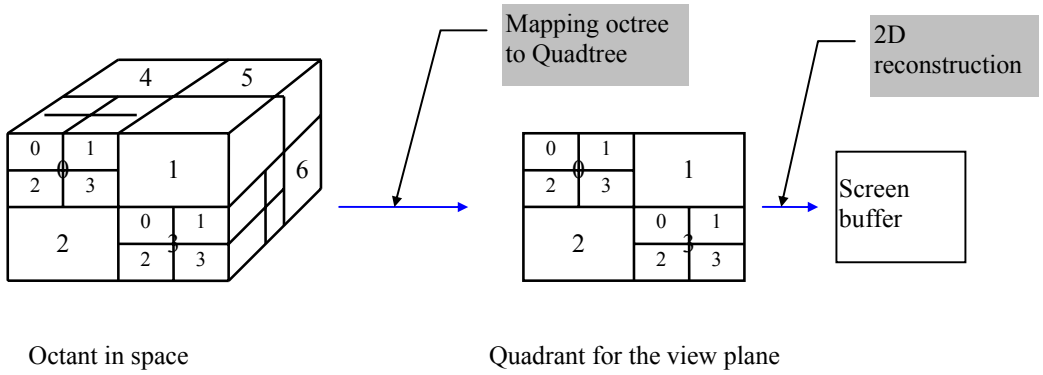


Figure 6.1: Octree display technique.

Recursive processing of octree nodes is demonstrated in the procedure listed in program 2.2 (chapter 2), which accepts an octree description and creates the quadtree representation for visible surfaces in the region. In most cases, both a front and a back octant must be considere in determining the correct color values for a quadrant. But if the front is homogeneously filled with some color, we do not process

tha back octant. For heterogeneous regions, the procedure is recursively called, passing a new as new arguments the child of the heterogeneous octant and a newly created quadtree node. If the front is empty, it is necessary only to process the child of the rear octant. Otherwise, two recursive calls are made, one for the rear octant and one for the front octant.

6.2.2 The application software :Parallel Viewer Software (PVS)

The application software (PVS) is designed for the medical staff to visualize medical organs from a certain perspective view and depth level with an orthogonal viewing direction. The desired view is obtained by rotations (rotate the volume organ by 90 degree to obtain lateral, top, and back views) and shifting the view plane to the location of the slice count in the orthogonal direction.

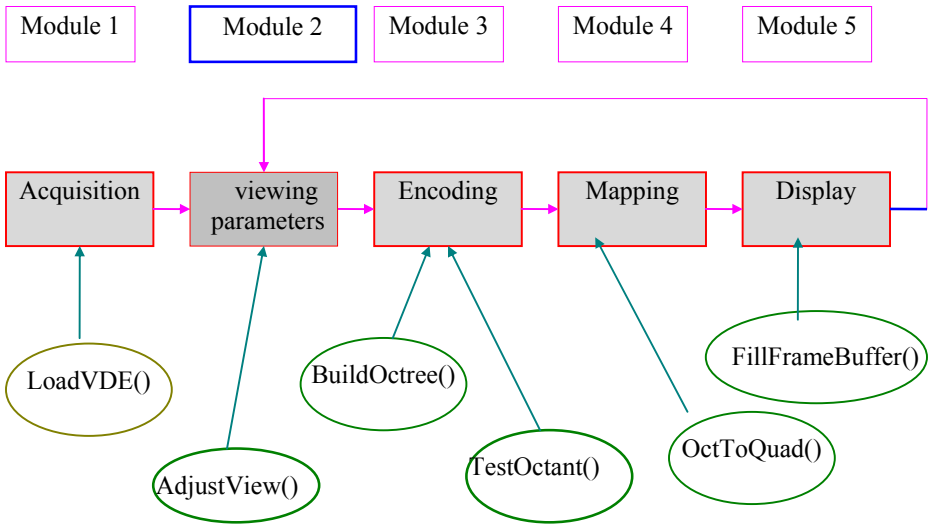


Figure 6.2: The application software which emploi the octree visualization algorithm to view the human organ according to selected viewing parameters.

The modules of the PVS software are interrelated through data paths following figure 6.2. The first module implements the data acquisition of the volume

using the ***LoadVDE()*** function, The second module perform viewing and depth adjustments to the volume data. Then, the octree algorithm pipe is performed through the following modules.the encoding module encodes the volume data into an octree representation using ***BuildOctree()*** and ***TestOctant()*** funstions, the mapping module maps the octree to the quadtree using ***OctToQuad()*** function, finally the display module displays the 2D reconstructed image using the ***FillFrameBuffer()*** function.

6.3 The parallel model

We consider the process farm model as shown in figure 6.3 and we design a parallel solution based on partitioning the volume data according to orthogonal volumes (ortvolume) as shown in figure 6.4. The master process **farmer** creates the worker processes **worker_i,** partitions the volume into orthogonal subvolumes of granularity G [refer to chapter 4, equation 4.3]. Then, it schedules them for execution by the worker processes. The orthogonal volume is composed of octants lying on the same orthogonal axis as shown in figure 6.4.

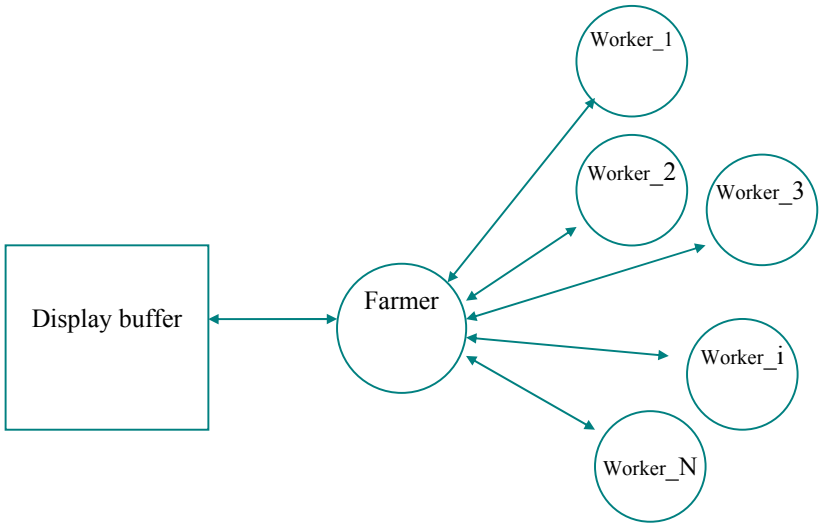


Figure 6.3: The process farm model

6.4 Data granularity, and task scheduling

The volume data is partitioned into orthogonal volumes called *ortvolumes*.The ortvolume is composed of all the voxels of the volume which are comprised along the orthogonal viewing direction and delimited by a 2D grid segment called gt. The orthogonal volume size is calculated as of: G*depth where G=gheight*gwidth.

The size of the *ortvolumes* is computed by the granularity parameters G and their location in volume space is identified by the task identifier *taskId* as explained in chapter 4.

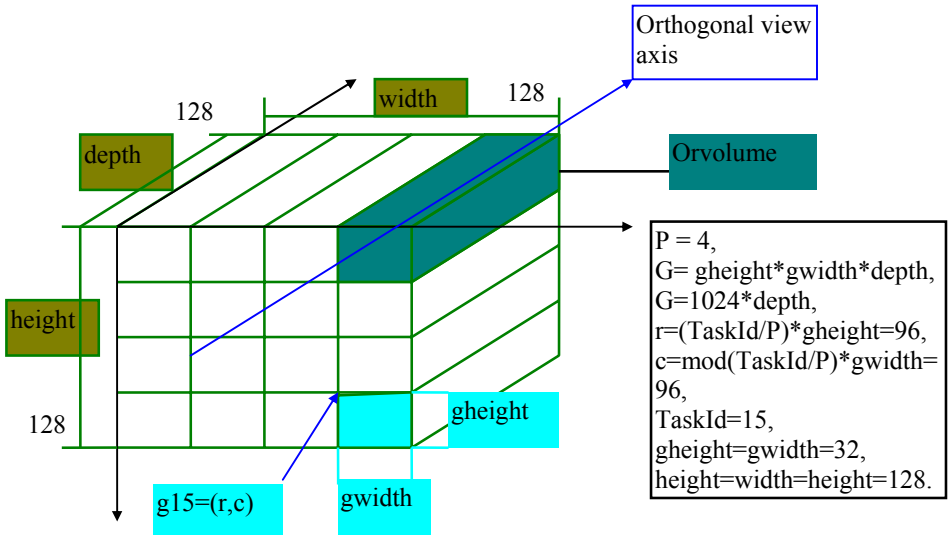


Figure 6.4: Partitioning the volume into orthogonal subvolumes,

The network is a fully connected network; all the worker processes can communicate with the master through channels using the *iserver* interface [inmos]. The synchronization between the master process and the worker processes is established through a protocol implemented by message packets: *taskmp, setupmp, ptaskmp*.

Each processor in the network is spawned with a worker process that executes received tasks (*taskmp*), and then sends the processes task (*ptaskmp*) to the master.

6.5 Messages and their protocols

The messages are formatted into packets *setupmp, taskmp, ptaskmp* according to the following protocol. Each field of the packet protocol is explained bellow:

1- setupmp: This packet is sent to the worker to initialize the appropriate workspace and transmission buffers.

```
typedef struct setupmp{
                int width; /*width*height*depth -> 3Dvolumesize*/
                int height; /*width*gwidth*height/gheight -> noftasks */
                int depth; /* The depth of the cube or orthogonal tunel*/
                int gwidth; /*gwidth*gheight -> 2Dgimagesize*/
                int gheight; /*gwidth*gheight*depth -> 3Dgvolumesize*/
                int gdepth;
                }Setupmp;
```

2- taskmp: This packet is sent to the worker, when the farmer schedules a certain task and it is composed of two fields, the taskId and the 3Dgvolumeptr, the 3D data of the grain volume pointer.

```
typedef struct taskmp{
                int taskId;             /*The Id of the 3Dgvolume*/
                unsigned char *3Dgvolumeptr;
                                /*The 3D data of grain volume pointer*/
                }Taskmp;
```

3- ptaskmp: This packet is received from a worker signaling an implied request for work and it returns two fields, the taskId of the processed grain and the 2Dimagptr corresponding to the 3Dgvolume.

```
typedef struct ptaskmp{
                int taskId;             /*The Id of the processed task*/
                unsigned char *2Dgimagptr;
                                /*The 2D image pointer of 3Dgvolume*/
                }Ptaskmp;
```

6.6 Process synchronisation of the process farm model

6.6.1 Process spawing and channel links CreatePlist(), CreateChannels()

```
main()
{
/*Initialization of globals(granularity:3Dgvolumesize, 2Dgimagesize, noftasks*/
Setupmp setup{32,32,32,4,4,4}; /*width, height, depth, gwidth, gheight, gdepth*/
noftasks = (width/gwidth)*(height/gheight); /*64*/

/*Connect the network with channels*/
  ChanListSize = nofworkers; ProcListSize = nofworkers + 1;
  chanlist = CreateChannels(chanlist, ChanListSize);
  plist = CreatePlist(plist,ProcListSize);
  plist[0] = ProcAlloc(farmer,40000,1,chanlist); /* 256 Kbytes for head*/
  for(i=1;i<ProcListSize; i++)
     {wid=i-1; wchan=chanlist[wid];
       plist[i]=ProcAlloc(worker,20000,2,wchan,wid); /*64 Kbytes for heap*/
     }
/*Spawn the processes on the network processors*/
  ProcParList(plist);
}
```

Note: The main program performs the following three tasks:

1- It initializes the theglobals:
granularity: 3Dgvolumesize and 2Dgimagesize,
the number of tasks using ***setup().***
2- Connect the network with channels,
It connect each worker with a channel ***chanlist[wid]***
3- Spawn the processes on the network using ***ProcParList(plist).***

6.6.1.1 CreateChannels() and CreatePlist()

The Connection of the network with channels (appch.c)

The processes of the network which are composed of the farmer and worker processes created by CreatePlist() function and connected with ProcAlloc(). The channels are created using the function CreateChannels() function. the implementation of this creation and connections is shown in program 6.6 bellow.

```
#include <stdlib.h>
#include <channel.h>
#include <process.h>
```

```
Channel ** CreateChannels(Channel *chanlist[], int ChanListSize)
{
      int Index;
      if((chanlist=malloc((ChanListSize + 1) * sizeof(Channel *))) ==NULL)
        abort();
      else
      {
        for(Index=0;Index<ChanListSize;Index++){
          chanlist[Index]=ChanAlloc();
          if (chanlist[Index]==NULL)
              abort();
        }
        chanlist[ChanListSize]=NULL;
      }
      return(chanlist);
}
Process ** CreatePlist(Process *plist[], int ProcListSize)
{
      if((plist=malloc((ProcListSize+1)*sizeof(Process *)))==NULL)
        abort();
      return(plist);
}
```

program 6.6: appch.c

6.6.2 The farmer process

```
farmer()
{
taskqueue = 0;
/* Serve the workers*/
for (i=0;i<(noftasks + nofworkers); i++)
{ identify a channel (ch);
  receive request (ptask); /* A worker request setup at the initial request*/
  if(taskqueue>=noftasks)/*tere is no more tasks: terminate workrs, eccessworker*/
     {SavePgrain();
       send task (task->taskId=STOP;
       break;                  /*goto receive next task*/
      }
  if(ptask->taskId!=INITIAL)
                  /*the first request for work (task) comes with no ptask*/
     { savePgrain();    /*save when it is a legal task*/
      }
/* Send a task*/
   {task->taskId=i;
    task->3Dgvolumeptr=3Dvolumeptr + i*3Dgvolumesize;
    send it through the channel;
    taskqueue++;
   }
 }
```

```
 }
```

6.6.3 The Worker process

The worker process is designed according to the following algorithm:

```
worker()
{
format ptask (with taskId=INITIAL) & send it to farmer;
for(i=0; i<noftasks;i++)
  {receive task;
   if (task->taskId == STOP) terminate;
   call the VisionPipe()
                /* The vision pipe is composed of the following functions:
                Build octree with BuildOctree();
                Map with OctToQuad();
                and reconstruct 2Dimage;
                */
   Send ptask (with task->2Dimageptr) to farmer process;
  }
}
```

6.7 Design of the functions of the Parallel Viewer Software (PVS)

6.7.1 Acquisition and Adjust View

6.7.1.1 Acquisition

The acquisition is performed using the LoadVDE function, This function fills a cube with data organ from a Volumetric Data Encoding (VDE) file.

char *array LoadVDE(VDEfile, *array)

Function: To adjust the viewing parameters, here we set for a cut view at depth
Arguments: VDEfile: a VDE format file
array: a three dimentional array
cube : boolean (returned value)

```
{
        space <-Allocate memory for header  palette and 3d object data
        if pace =TRUE
        identify the number of slices N
        for (i= 0 to N)
        {
                read slice i
                promote image to 24 bit per pixel
                interpolate
                filter
                convert to 8 bits per pixel
                put slice i in cube
        }

                }
       else
                cube <- FALSE;
}
return cube;
```

6.7.1.2 BOOL AdjustView(Depth)

Function: To adjust the viewing parameters, here we set for a cut view at depth
Arguments: Depth: Specify the display depth
View: a returned boolean , TRUE: valid, FALSE: not valid

```
{
        If Depth is within the bounding limits of the cube
                {
                View <- TRUE;
                determine the bounding limits of the new cube by shifting the front
face
                 to the viewing plane to Depth.
                call BuildOctree function with the back shifted slices set to void
                }
        else
                view <- FALSE;
}
return view;
}
```

6.7.2 The octree visualization algorithm (Vision pipe)

The vision pipe is composed of three modules : encoding, mapping, and display

6.7.2.1 Encoding: The octree encoding is performed with the following functions: BuildOctree(), InitOctNode(), TestOctant().

BOOL BuildOctree(Octreeptr Root, char *Array)

Function: *The BuildOctree() function builds the octree of the cube*

Arguments: ***Root:*** a node of type Octreeptr

Array: a pointer to the cube that holds the 3D (volume data:organ)

Global variables: ***Pass:*** a boolean variable to initialize the root node fields by giving the upper and lower limits of the entire universe and setting its state to heterogeneous.

Locals variables: ***Done:*** a boolean returning the state of the function

NodeIndex: an index to the parent nodes

KidIndex: and index to the kid nodes

```
{
    test if the entire universe node is initialized to pass the initialization portion
    if Pass = FALSE
    {
        initialize the entire universe node's fields
        Root->Lower<-[0,0,0];
        Root->Upper<-[128,128,128];
        Root->Hmg<-Color <-NOTUSED;
        Pass <- TRUE;
    }
    if Pass=TRUE
    {
    build the eight octants and link them to the entire universe
    for NodeIndex <-0 to 7
    {
        Root->Octant[NodeIndex] <- new Octree;
        call InitOctNode function to initialize the octant's fields
        InitOctNode(Root->Octant[NodeIndex], NodeIndex) ;
        call TestOctant function to test the state of the octant
        Root->Octant[NodeIndex]->Hmg
            <-TestOctant(Root->Octant[NodeIndex], Array);
        see if the octant is heterogeneous, void or homogeneous
        if Root->Octant[NodeIndex] -> Hmg=H_FALSE    /*heterogeneous*/
            then call BuildOctree function recursively
            Done<-BuildOctree(Root->Octant[NodeIndex], Array);
        else
            for KidIndex <- 0 to 7
                null all the kids of the octant
                Root<-Octant[NodeIndex]->Octant[KidIndex]<-NULL;
    }
}
```

```
    retrun Done;
} /* end of BuildOctree()*/
```

BOOL InitOctNode(Octreeptr Root, int Index)

Function: Initialize the Lower and Upper fields of the octree node

Arguments: ***Root***: node of type Octreeptr

Index: index to the node

We define as a notation:

X: (Root->Lower.x + (Upper.x - Root->Lower.x)/2

Y: (Root->Lower.y + (Upper.y - Root->Lower.y)/2

Z: (Root->Upper.z + (Upper.z - Root ->Lower.z)/2

```
{
    test the Index value
    switch Index
        case 0:
            Root->Octant[0]->Lower.x <- Root->Lower.x;
            Root->Octant[0]->Lower.y <- Root->Lower.y;
            Root->Octant[0]->Lower.z <- Root->Lower.z;
            Root->Octant[0]->Upper.x <- X;
            Root->Octant[0]->Lower.y <- Y;
            Root->Octant[0]->Lower.z <- Z;
            break;
        case 1:
            Root->Octant[1]->Lower.x <- X;
            Root->Octant[1]->Lower.y <- Root->Lower.y;
            Root->Octant[1]->Lower.z <- Root->Lower.z;
            Root->Octant[1]->Upper.x <- Root->Lower.x;
            Root->Octant[1]->Lower.y <- Y;
            Root->Octant[1]->Lower.z <- Z;
            break;
        case 2:
            Root->Octant[2]->Lower.x <- X;
            Root->Octant[2]->Lower.y <- Y;
            Root->Octant[2]->Lower.z <- Root->Lower.z;
            Root->Octant[2]->Upper.x <- Root->Lower.x;
            Root->Octant[2]->Lower.y <- Root->Lower.y;
            Root->Octant[2]->Lower.z <- Z;
            break;
        case 3:
            Root->Octant[3]->Lower.x <- Root->Lower.x;
            Root->Octant[3]->Lower.y <- Y;
            Root->Octant[3]->Lower.z <- Root->Lower.z;
            Root->Octant[3]->Upper.x <- X;
            Root->Octant[3]->Lower.y <- Root->Lower.y;
            Root->Octant[3]->Lower.z <- Z;
            break;
        case 4:
```

```
                    Root->Octant[4]->Lower.x <- Root->Lower.x;
                    Root->Octant[4]->Lower.y <- Root->Lower.y;
                    Root->Octant[4]->Lower.z <- Z;
                    Root->Octant[4]->Upper.x <- X;
                    Root->Octant[4]->Lower.y <- Y;
                    Root->Octant[4]->Lower.z <- Root->Lower.z;
                    break;
            case 5:
                    Root->Octant[5]->Lower.x <- X;
                    Root->Octant[5]->Lower.y <- Root->Lower.y;
                    Root->Octant[5]->Lower.z <- Z;
                    Root->Octant[5]->Upper.x <- Root->Lower.x;
                    Root->Octant[5]->Lower.y <- Y;
                    Root->Octant[5]->Lower.z <- Root->Lower.z;
                    break;
            case 6:
                    Root->Octant[6]->Lower.x <- X;
                    Root->Octant[6]->Lower.y <- Y;
                    Root->Octant[6]->Lower.z <- Z;
                    Root->Octant[6]->Upper.x <- Root->Lower.x;
                    Root->Octant[6]->Lower.y <- Root->Lower.y;
                    Root->Octant[6]->Lower.z <- Root->Lower.z;
                    break;
            case 7:
                    Root->Octant[7]->Lower.x <- Root->Lower.x;
                    Root->Octant[7]->Lower.y <- Y;
                    Root->Octant[7]->Lower.z <- Z;
                    Root->Octant[7]->Upper.x <- X;
                    Root->Octant[7]->Lower.y <- Root->Lower.y;
                    Root->Octant[7]->Lower.z <- Root->Lower.z;
                    break;
        return;
} /*end of initOctNode*/
```

int TestOctant(Octreeptr Octptr, char *Array)

function: test the state of the octant if void, homogeneous, or heterogeneous

Arguments: ***Octptr***: a node of type Octreeptr
Array: a pointer to the cube that holds the 3D organ.

Locals: ***Hmg***: int to return the state of the octant
Turn: a control variable used to stop the test operation and to return the state of the octant.
Offset: a used to calculate the position of the point in the 3D cube which is an array organized as three dimentional array.
Color: A dummy variable to hold the previous color value.
DepthIndex: an index along the Z axis
HeighIndex: an index along the Y axis
WidthIndex: an index along the X axis

```
{
        test if the maximum partitioning is reached (idealy the graphical resolution)
        if Octptr ->Lower = Octptr ->Upper
        {
                calculate the offset to determine the position of the voxel in the cube
                Offset<-128 * 128 * Octptr -> Lower.z + 128 * Octptr -> Lower.y +
                                Octptr->Lower.x;
                if Array[Offset]!=0
                {
                        the voxel has a unique color
                        Hmg<-H_TRUE;
                        Octptr->Color <- Array[Offset];
                }
                else
                {
                        the voxel has no color (empty voxel)
                        Hmg<-H_EMPTY;
                        Octptr->Color <- 0;
                }
        }
        else
        {
                the graphical resolution is not reached, then we go on with the testing
                of all the voxels
                for DepthIndex<-Octptr->Lower.z to Octptr->Upper.z
                  for HeightIndex<-Octptr->Lower.y to Octptr->Upper.y
                    for WidthIndex<-Octptr->Lower.x to Octptr->Upper.x
                    {
                        Offset<-128 * 128 * DepthIndex + 128 * HeightIndex +
                                WidthIndex;
                        test the voxel if it is not zero to stop testing for empty state
                        if Color<-Array[Offset]!=0
                                Turn <- 1;
                        break the three nested loops
                    }
        test if the empty state is not valid
        if Turn = 1
        {
                set three nested loops in order to test all the voxels
                for DepthIndex<-Octptr->Lower.z to Octptr->Upper.z
                  for HeightIndex<-Octptr->Lower.y to Octptr->Upper.y
                    for WIdthIndex<-Octptr->LOwer.x to Octptr->Upper.x
                    {
                        Offset <- 128 * 128 * DepthIndex + 128 * HeightIndex +
                                        WidthIndex;
                        test if the voxels have the same color
                        if Array[Offset]!= Color
```

```
                        the voxels do not have the same color
                        Turn<- 2;
                    break the three nested loops
                }
            }
            switch Turn
            case 0:
                    Hmg<- H_EMPTY;
                    Octptr->Color <- 0;
                    break;
            case 1:
                    Hmg<- H_TRUE;
                    Octptr->Color<-Color;
                    break;
            case 2:
                    Hmg<-H_FALSE;
                    Octptr->Color<-NOTUSED;
                    break;
            }
            return Hmg;
    } /* end of TestOctant */
```

6.7.2.2 Mapping

The octree mapping into quadtree is performed with the following functions: **OctToQuad(), BuildQuadNode(), DestroyQuadKids(), InitQuadNode().**

BOOL OctToQuad(Octreeptr Ocroot, Quadtreeptr Quadroot)
Function: to build the map of the octree to quadtree
Arguments:
Octroot: node of type Octreeptr
Quadroot: node of type Quadtreeptr
Locals:
Done: The Boolean return value of the function
NodeIndex: an index to node

```
{
    for NodeIndex <- 0  to 3
    {
        test the state of the front nodes in the octree
        switch Octroot-> Octant[NodeIndex] ->Hmg
        {
        case  H_TRUE:
            Quadroot->quadrant[NodeIndex]->Hmg<-H_TRUE;
            Quadroot->quadrant[NodeIndex]->Color <-
                        Octroot->Octant[NodeIndex]->Color;
            null the kids of the node
            DestroyQuadKids(Quadroot->quadrant[NodeIndex]);
            break;
```

```
case H_EMPTY:
        test the back nodes in the octree
        switch Octroot->Octant[NodeIndex+4]->Hmg
          {
          case H_TRUE:
                Quadroot->quadrant[NodeIndex]->Hmg<-H_TRUE;
                Quadroot->quadrant[NodeIndex ]->Color<-Octroot
                ->Octant[NodeIndex+4]->Color;
                DestroyQuadKids(Quadroot->quadrant[NodeIndex]);
                break;
          case H_EMPTY:
                Quadroot->quadrant[NodeIndex]->Hmg<-H_EMPTY;
                Quadroot->quadrant[NodeIndex]->Color<-0;
                break;
          default:
                Quadroot->quadrant[NodeIndex]->Hmg<-H_FALSE;
                Quadroot->quadrant[NodeIndex]->Color<- NOTUSED;
                build the kids of the node in the quadtree
                Done <- BuildQuadNodes(
                                Quadroot->quadrant[NodeIndex]);


                  if Done != FALSE
                  map the octree with the back nodes to the quadtree
                  Done<-OctToQuad(Octroot->Octant[NodeIndex+4],
                          Quadroot->quadrant[NodeIndex]);
          break;
          }
        break;
        default:
        Quadroot->quadrant[NodeIndex]->Hmg <- H_FALSE;
        Quadroot->quadrant[NodeIndex]->Color <- NOTUSED;
        build the kids of the node in the quadtree
        Done<-BuildQuadNode(Quadroot->quadrant[NodeIndex]);
        if Done!= FALSE
          map the octree with the back nodes to the quadtree
          Done<-OctToQuad(Octroot->Octant[NodeIndex+4],
                        Quadroot->quadrant[NodeIndex]);
        if Done!= FALSE
          map the octree with the front nodes to the quadtree
          Done<-OctToQuad(Octroot->Octant[NodeIndex],
                        Quadroot->quadrant[NodeIndex]);
          }
        break;
        }
    }
    return Done;
```

```
} /* end of OctToQuad */
```

void BuildQuadNode(Quadtreeptr Quadroot)
Function: The following function builds the kids of a quadtree node
Arguments: **Quadroot**: a node of type Quadtreeptr
Locals: **Done**: a boolean value to be returned by the function
NodeIndex: an index to a node

```
{
    Done <- TRUE;
    for NodeIndex <- 0 to 3
    {
        if(Quadroot->quadrant[NodeIndex] <- new Quadtree) = NULL
        {
            Done <-FALSE;
            break;
        }
        if Done != FALSE
        {
            for NodeIndex <- 0 to 3
            {
                null the kids of the node
                DestroyQuadKids(Quadroot->quadrant[NodeIndex]=;
                Initialize the node field
                InitQuadNodes(Quadroot, NodeIndex);
            }
        }
        return Done;
    } /* end of BuildQuadNode*/
```

void DestroyQuadKids(Quadtreeptr Quadroot)
Function: The following function nulls the kids of the quadtree node.
Arguments: **Quadroot**: a node of type Quadtreeptr
locals: **NodeIndex**: an index to a node

```
{
    for NodeIndex<- 0 to 3
        Quadroot->quadrant[NodeIndex]<- NULL;
} /*end of DestroyQuadKids*/
```

void InitQuadNode(Quadtreeptr Quadroot, int Index)
Function: It initializes the Upper and Lower fields of the new quadtree node
Arguments: **Quadroot:** a node of type Quadtreeptr
Index: index to a node
we define as a notation:

```
    MX:(Quadrant->Upper.x - Quadroot->Lower.x)/2;
    MY:(Quadroot ->Upper.y - Quadroot->Lower.y)/2;
```

```
{
        test Index value.
        switch Index
        {
                case 0:
                        Quadroot->quadrant[0]->Lower.x  <- Quadroot ->Lower.x;
                        Quadroot->quadrant[0]->Lower.y  <- Quadroot ->Lower.y;
                        Quadroot->quadrant[0]->Upper.x  <- Quadroot->Upper.x;
                        Quadroot->quadrant[0]->Upper.y  <- Quadroot->Upper.y;
                        break;
                case 1:
                        Quadroot->quadrant[1]->Lower.x <- MX;
                        Quadroot->quadrant[1]->Lower.y <- Quadroot -> Lower.y;
                        Quadroot->quadrant[1]->Upper.x <- Quadroot -> Upper.x;
                        Quadroot->quadrant[1]->Lower.y <- MY;
                case 2:
                        Quadroot->quadrant[2]->Lower.x <- MX;
                        Quadroot->quadrant[2]->Lower.y <- MY;
                        Quadroot->quadrant[2]->Upper.x <- Quadroot -> Upper.x;
                        Quadroot->quadrant[2]->Lower.y <- Quadroot -> Upper.y;
                case 3:
                        Quadroot->quadrant[3]->Lower.x <- Quadroot ->Lower.x;
                        Quadroot->quadrant[3]->Lower.y <- MY;
                        Quadroot->quadrant[3]->Upper.x <- MX;
                        Quadroot->quadrant[3]->Lower.y <- Quadroot -> Upper.y;
                        break;
                }
                return;
        } /* end of DestroyQuadKids*/
```

6.7.2.3 Display: The display is performed by the functions: **TraverseQuadTree()** and **FillBuffer()**.

TraverseQuadtree(Quadroot, Buffer)

function: to traverse the quadtree and to construct the 2D image.

Arguments: ***Quadroot***: a node of type Quadtreeptr
Buffer an array to hold the 2D image for display

Locals: ***Offset***: to point to the desired pixel in the array
NodeIndex: an index to the quadtree node
WidthIndex: an index along the x axis
HeightIndex: an index along the y axis.

```
{
for NodeIndex <60 to 3
        if Quadroot ->quadrant[NodeIndex] != NULL
                TraverseQuadTree(Quadroot->quadrant[NodeIndex], Buffer);
        else
        {
```

```
                for HeightIndex<-Quadroot->Lower.y to Quadroot->Upper.y
                        for WidthIndex<- Quadroot->Lower.x to Quadroot->Upper.x
                        {
                        Offset <- 128*HeightIndex + WIdthIndex;
                        Buffer[Offset]<-Quadroot->Color;
                        }
        }
} /* end of TraverseQuadTree*/
```

FillBuffer()

Function: This function fills the display frame buffer with 2D image obtained from the Traverse three. It is assumed to be performed by the farmer process.

```
{
        Turn the screen to graphic mode
        for (Imageptr <-0 to Imageptr+width)
                for (imageptr <- 0 to length)
                        {
                        read data from 2D image array
                        display on the screen using putpixel()
                        }
}
```

6.8 Implementation

In this section we present the implementation of the parallel viewer software (PVS).

6.7.1 The PVS (Parallel viewer Software) version 3.

```
/*************************************************************************/
/* Filename:    c:\ictools\examples\master\v3\vc2a.c                     */
/*  Title: Parallel Viewer Software (PVS)                                */
/*  Abstract: The Parallel Viewer Software (PVS) is designed and implemented under the */
/*              the ANSIC parallel compiler for Transputer networks. It visualizes any 3D */
/*              data. The data should be encoded into a format called Volume Data Encoding */
/*              (VDE) presented in chapter 2.                            */
/*              The PVS software is a concurrent software designed with the  process farmer */
/*               parallel model. The worker processes spawned on the network receive tasks */
/*               (3D sub-volumes ) compute the vision pipe and transmits back to the farmer */
/*           PVS:                                                        */
/*           This work has started from the original programs  vc0.c which is stored in the */
/*           c:\ictools\examples\project\master\v1\vc1.c                 */
/*           The version 2 & 2a are a rational change from vc1           */
/*           Here we are  followed an implementation corresponding       */
/*           to a design presented in v2\vc2.doc.                        */
/*           - synchronisation through message packets                   */
/*           - Global variables                                          */
/*           PVS is a software to read VDE file formats, Adjust the viewing parameters and */
/*           displays the 3D view.                                       */
/* Summary:     This work demonstrate the parallelization of the        */
/*           octree volume visualization algorithm. using the Octree and Quadtree encoding and */
/*           visualization techniques.                                   */
/*           Data partitioning follows orthogonal tunnels of granularity */
/*           defined in setup {32,32,32,4,4,4} structure                 */
/*           The farmer process sends a message packet of type           */
/*           Taskmp to the worker processes. This latter generate        */
/*           the sub quatree and reconstruct its                         */
/*           sub 2D image and send it back to the farmer process         */
/*           formatted in a message packet of type Ptaskmp               */
/*           synchronization has been implemented using                  */
/*           the algorithm designed in vc2.doc                           */
/* Hauthor:     Mr. Bouklachi abbes,                                     */
/*           Head of the Microprocessor research team,                   */
/*           Post-graduate department,                                   */
/* Institute:   National Institute of Electricity and                    */
/*           Electronics, (INELEC) , Boumerdes                           */
/* Date:        July,11, 1999                                           */
/* Language:    ANSI C parallel programming language                    */
/*                                                                       */
/* libraries:  - graphics.c : initgraph(), drawpnt(), AlphaMode(), ConvtoGrayScale(), */
/*                            DisplayImage(), ReadImage(), LoadVDE()     */
/*             -  vision.c:   BuildOctree(), TestOctant(), InitOctNode(), */
/*                            InitQuadNode(), BuildQuadNode(),OctToQuad(), DestroyQuadKids */
/*             - netorw.c:     CreateChannels(), CreatePlist()           */
/*             - dplb.c:       Gaussp(), SavePgrain()                    */
/* Linkage:     ilink pvs.tco graphics.tco dplb.tco vision.tco  network.tco */
/*              /f startup.lnk /t425                                     */
/*************************************************************************/
```

```
/*   .....PVS.....*/
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<process.h>
#include<channel.h>      /*Process and Channel*/
#include <float.h>
#include <math.h>
#include <mathf.h>
#include <dos.h>
#include "graphic2.h"    /*Initgraph(),ReadImage(),BuildCube(),LoadVDE() …*/
#include "vc2.h"         /* Globals varaibles ,function prototypes ...*/
#include "appch2.h"      /* Channel prototypes*/

/*   .....farmer process.....*/

static void farmer(Process *p,Channel *chanlist[])
{
 int i,k,ch,taskqueue;
 int noftasks,nofworkers;
 unsigned char *3Dvolumeptr;    /*points to the volume data*/
 unsigned char *2Dimageptr;     /*points to the reconstructed image*/

 Ptaskmp rptask;   /* local to farmer:rptsak->tsakId,rptask->2Dgimageptr*/
 Taskmp ttask;     /* local to farmer:ttask->taskId, ttask->3Dgvolumeptr*/

 extern Setup gran;    /*global  set in main*/
 extern Network net;    /*global set in main*/

 2Dgimagesize=gran->gwidth*gran->gheight;
 3Dvolumesize=gran->width*gran->height*setup->depth;
 3Dgvolumesize=2Dgimagesize*gran->depth;
 2Dimagesize=gran->width*gran->height;
 ptaskmsize=2Dgimagesize+sizeof(int);
 taskmsize=3Dgvolumesize+sizeof(int);
 noftasks=(gran->width/gran->gwidth)*(gran->height/gran->gheight);
 nofworkers=net->nofworkers;
 p=p;

  /*-1 Allocate space for 2Dgimage at rptask->2Dgimageptr*/
if((rptask->2Dgimageptr=calloc(rptask->2Dgimagesize,sizeof(char)))==NULL)            /* ... */
 abort();
  /*-2 Allocate a space of 2Dimagesize  at 2Dimageptr */
if((2Dimageptr=calloc(2Dimagesize,sizeof(char)))==NULL)
 abort();
  /*-3 Allocate a space of 3Dvolumesize at 3Dvolumeptr  */
if((3Dvolumeptr=calloc(3Dvolumesize,sizeof(char)))==NULL)
 abort();

  /*...          Build the cube                    ...*/
BuildCube(&3Dvolumeptr,gran);      /*graphic0.c::*/
```

```
taskqueue =0;
/* ..................................................Serve the workers ..........................................................*/
  for(i=0;i<(noftasks+nofworkers);i++)
  {      /*Block on  channels chanlist*/
            ch=ProcAltList(chanlist);                /*identify a channel ch*/
            ChanIn(chanlist[ch],rptask,ptaskmsize);   /*receive request rptask */

            if(taskqueue >= noftasks)             /*There is no more tasks:*/
              {                                  /*Shutdown workers,eccessworker*/
                SavePgrain(rptask->2Dgimageptr,2Dimageptr,gran);
                ttask->taskId = SHUTDOWN;
                Chanout(chanlist[ch],ttask,taskmsize);
                break();                         /*Receive next request*/
              }

            if(rptask->taskId != INITIAL)         /*Save when rptask is legal */
                SavePgrain(rptask->2Dgimageptr,2Dimageptr,gran);
            /*Send a ttask  */
            ttask->taskId = i;
            ttask->3Dgvolumeptr=3Dvolumeptr + i*3Dgvolumesize;
            ChanOut[chanlist[ch],ttask,taskmsize);
            taskqueue ++;
            /*End send ttask */
  }


/*..... Display original image and processed image.....*/
k=1;
DisplayImage(2Dimageptr,gran,k);
/*.....................................................*/


free(2Dgimageptr);         /* free memory  */
free(3Dgvolumeptr);
free(3Dvolumeptr);
}
```

```
/*.....*/
/*                                    worker process                                    */
/*.....*/
static void worker(Process *p,Channel *wchan, int wid)
{
 union REGS inregs;

 Ptaskmp tptask;   /* local to worker:tptask->taskId,tptask->2Dgimageptr*/
 Taskmp rtask;     /* local to worker:rtask->taskId, rtask->3Dgvolumeptr*/

 extern Setup gran;    /*global set  in main*/
 p = p;

 2Dgimagesize=gran->gwidth*gran->gheight;
 3Dgvolumesize=2Dgimagesize*gran->depth;
 ptaskmsize=2Dgimagesize+sizeof(int);
 taskmsize=3Dgvolumesize+sizeof(int);

  /*-1 Allocate space for 2Dgimage at tptask->2Dgimageptr*/
 if((tptask->2Dgimageptr=calloc(2Dgimagesize,sizeof(char)))==NULL)              /* ... */
  {printf("Not enough memory for 2Dgvolumesize wid:%d\n",wid);exit(1);}
  /*-2 Allocate a space of 3Dgvolumesize at rtask->3Dgvolumeptr  */
 if((rtask->3Dgvolumeptr=calloc(3Dgvolumesize,sizeof(char)))==NULL)
  {printf("Not enough memory for 3Dgvolumesize wid:%d\n",wid);exit(1);}
                        /*.....*/
 /*1 format tptask (with tptask->taskId=INITIAL) &send it to farmer*/
 tptask->taskId=INITIAL;
 ChanOut(wchan,tptask,ptaskmsize);/*tptask->2Dgimageptr is garbedge*/
 do{
          /* -1 Receive rtask*/
          ChanIn(wchan,rtask,taskmsize);   /*receive rtask in rtask*/
          if(rtask->taskId==SHUTDOWN)      /*if rtask->taskId==SHUTDOWN */
            break();                    /*terminate*/
          printf("W%d received task <---:%d\n",wid,rtask->taskId);

          /* -2 Visualization pipe*/
          printf("worker %d calls visionpipe(rtask=%d) \n",wid,rtask->taskId);

          VisionPipe(gran, rtask, tptask);   /*vision::VisionPipe()*/
          /*Setup gran,Taskmp rtask,Ptaskmp tptask*/

          /* -3 Send tptask*/
          tptask->taskId = rtask->taskId;
          /*tptask->2Dgimageptr=tptask->2Dgimageptr;*/
          ChanOut[wchan,tptask,ptaskmsize);
          printf("W%d send result (request work) ---> ",wid);
          /* printf(".");  */
            inregs.x.cx=rtask->taskId;                  inregs.x.dx=100;
            inregs.h.al=wid;   /*color of worker*/ inregs.h.ah=0x0c; /*Write pixel*/
            int86(0x10,&inregs,&inregs);
}while(TRUE};
 free(tptask->2Dgimageptr);  free(rtask->3Dgvolumeptr);

}
```

```
/*.....main .....*/
int main(void)    /* The main program: for octree visualization  */
{
            int wid,Index,ChanListSize,ProcListSize;
            Process **plist=NULL;
            Channel **chanlist=NULL,*wchan;
            initgraph();

            /* Initialition of global variables defined a vc.h             */
            /*.....*/
 Setup gran{32,32,32,4,4,4};   /*global  set in main*/
 Network net{2};                  /*global nofworkers set in main*/

 2Dgimagesize=gran->gwidth*gran->gheight;
 3Dvolumesize=gran->width*gran->height*gran->depth;
 3Dgvolumesize=2Dgimagesize*gran->depth;
 2Dimagesize=gran->width*gran->height;
 ptaskmsize=2Dgimagesize+sizeof(int);
 taskmsize=3Dgvolumesize+sizeof(int);
 nofworkers=net->nofworkers;
 nofgrains=(gran->width/gran->gwidth)*
             (gran->height/gran->gheight); /*nofgrains=8*8=64               */
 ChanListSize=net->nofworkers;          /*(nofworkers) list of channels    */
 ProcListSize=net->nofworkers+1;        /*(farmer+nofworkers)list of procs */

            /*.....*/
            chanlist=CreateChannels(chanlist,ChanListSize);
            plist=CreatePlist(plist, ProcListSize);
            plist[0]=ProcAlloc(farmer,40000,1,chanlist); /*256Kbytes for heap...*/
            if (plist[0]==NULL)
             abort();
            for(Index=1;Index<ProcListSize;Index++){
             wid=Index-1;
             wchan=chanlist[wid];
             plist[Index]=ProcAlloc(worker,20000,2,wchan,wid); /*20000 heap*/
             if (plist[Index]==NULL)
               abort();
            }
            plist[ProcListSize]=NULL;

            ProcParList(plist);             /* Start the system     ... */

            for(Index=0;Index<ProcListSize;Index++)
                        ProcAllocClean(plist[Index]);
            tfinal=ProcTime();
            comptime[loop]=(int) ProcTimeMinus(tfinal,tinit)/1000;
           /* printf("loop.nofgrains.nofworkers.comptime:%d %d %d %d\n",
                        loop,nofgrains,nofw[loop],comptime[loop]); */
            fprintf(fp,"loop.nofgrains.nofworkers.comptime:%d %d %d %d\n",
                        loop,nofgrains,nofw[loop],comptime[loop]);

    AlphaMode();
    fclose(fp);
    printf("The end.....\n");
}
```

6.8.2 Global and functions prototypes (pvs.h)

```
/*.....................................................................*/
/*              Global variables                                       */
/*.....................................................................*/
#define INITIAL -1
#define SHUTDOWN -1


typedef struct setup{
                int width;
                int height;
                int depth;
                int gwidth;
                int gheight;
                int gdepth;
                }Setup;
typedef struct network{
                 int nofworkers;
                 }Network;
typedef struct taskmp{
                 int taskId;
                 unsigned char *3Dgvolumeptr;
                 }Taskmp;
typedef struct ptaskmp{
                 int taskId;
                 unsigned char *2Dgimageptr;
                 }Ptaskmp;

void VisionPipe(void);
void SavePgrain(                /*The farmer saves rptask into 2Dimageptr*/
            unsigned char *,  /*rptask->2Dgimageptr */
            unsigned char *,  /*2Dimageptr */
            Setup           /*grain->width,grain->height,grain->depth*/
            );
```

Octree.h

```
/*.....................................................................*/
/*...          Octree library (octree.h)                        ...*/
/*.....................................................................*/
#define H_EMPTY 0
#define H_FALSE 1
#define H_TRUE 2
#define FALSE 0
#define TRUE 1
static exist=FALSE;
static mapped=FALSE;
static Pass=FALSE;
static draw=FALSE;

typedef struct point_3D{
                  int x;
                  int y;
                  int z;
```

```
                }Point3D;
typedef struct point_2D{
                int x;
                int y;
                }point2D;
typedef struct octree{
                Point3D Lower;
                Point3D Upper;
                int Hmg;
                int Color;
                struct octree *Octant[8];
                }Octree;

typedef struct quadtree{
                Point2D Lower;
                Point2D Upper;
                int Hmg;
                int Color;
                struct quadtree *quadrant[4];
                }Quadtree;

typedef Octree *Octreeptr;
typedef Quadtree *Quadtreeptr;
char *MyArray,*Buffer;
int Offset;
BOOL exist,mapped,constr,draw; BOOL Pass;
Octreeptr OctRoot;
Quadtreeptr QuadRoot;
BOOL BuildOctree(Octreeptr root,char *Array);
int TestOctant(Octreeptr Octptr, char *Array);
BOOL OcToQuad(Octreeptr Octroot, Quadtreeptr Quadroot);
BOOL BuilQuadNodes(Quadtreeptr Quadroot);
void InitQuadRoot(Quadtreeptr Quadroot);
void TraverseQuadTree(Quadtreeptr Quadroot,char *Array);
```

graphic2.h

```
/*.....................................................................*/
/*                        graphic2.h                                   */
/*.....................................................................*/
void drawpnt(int *, int *, int *);
void initgraph(void);
void AlphaMode(void);
void ConvtoGrayScale(void);
void DisplayImage(unsigned char *,int ,int ,int );
void ReadImage(unsigned char **,int,int);
void BuildCube(unsigned char **,Setup);
            /*  unsigned char *cubearray,
              int gran->depth,gran->height,gran->width,
              int gran->gdepth,int gran->gheight,int gran->gwidth.*/
void InitQuadImage(unsined char **,int,int);
           /*unsigned char *qimageptr,
              int width, int height*/
char *array LoadVDE(VDEfile, *array);
```

6.8.3 Graphics library

The graphics library is composed of the necessary functions which interface the transputer board to the VGA graphics cards. These functions have been written using BIOS input output services. Here are the designed fuctions: initgraph(), drawpnt(), AlphaMode(), ContoGrayScale(), DisplayImage(), and LoadVDE().

6.8.4 network library

The network libray is composed of two functions CreateChannels() and CreatePlist(). The CreateChannels() creates the necessary channels to connect the processes for communication and the CreatPlist() creates the processes specified in the list.

6.8.5 Data partitioning library : SavePgrain()

```
/*****************************************************/
/*Hauthor:Abbes Bouklachi                                ... */
/*Library:dplb799.c: data partitioning library.          ... */
/* SavePgrain(),                                         ... */
/*void SavePgrain(rptask, fiomemptr,setup)               ... */
/*.....*/
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <float.h>
#include <math.h>
#include <mathf.h>
#define pi 3.141592654
#define round(x) (int) ((x)+0.5)

/*.................SavePgrain().....................*/

void SavePgrain(rptask, 2Dimageptr, setup)
unsigned char *2Dimageptr;
Ptaskmp rptask;
Setup setup;
int width,gwidth,height,gheight;
{
 int i,j,txc,tyc;
 unsigned char *pimage=2Dimageptr;
 txc=(int) rptask->taskId/(setup->width/setup->gwidth);
 tyc= rptask->taskId%(setup->height/setup->gheight);
 for(i=0;i<gheight;i++)
```

```
  for(j=0;j<gwidth;j++){
    pimage[tyc*setup->gwidth+
              txc*setup->gheight*setup->width+j+
              i*setup->width] = (char) 2Dimagptr[j+i*setup->gwidth+1];
   }
}
```

6.8.6 Vision Library

The vision library is composed of following functions: BuildOctree(), TestOctant(), InitOctNode(), InitQuadNode(), BuildQuadNode(), OctToQuad(), and DestroyQuadKids().

6.9 Performance of the parallel solution

A simulation work has been done over windows95. A software has been written in VISUAL C++ to implement the PVS design with four processors.

The performance we have obtained is a significant decrease in the execution time compared to the sequential program.

We have gathered the timing of the executuion time of the worker processes and the master processs in the table 6.12.

Table 6.12: recorded timing for various granularities

P: Partition parameter	**G:** Grain size	Number of grains in Vision tunnel	**R:** Running time for one G	**C:** Communication time	**S** Speed up
1	128*128*128	1	284.069		1
2	64*64*64	2	32.239		.11
4	32*32*32	4	4.018		0.01
8	16*16*16	8	0.566		0.001
16	8*8*8	16	0.301		0.0007
32	4*4*4	32	0.209		0.0007

The performance of this parallel solution is a function of the granularity G and the number of processes N. we notice from table 6.12 that each time we decrease the grain size (G) or the sub-volume dimensions, we obtain a reduced running time R.

When the partitioning parameters is 1 it means that we haven't sub-divided the volume and the obtained timing represent the sequential running time of the algorithm (284.069 ms).

The optimum grainsize has been found at (G=16) which give the best performance for a medical application. If the partitioning goes beyond that optimum value, the communication overhead will get greater than the running time.

6.10 Conclusion

What we have obtained is very encouraging, and the author whish to carry further work to design and implement medical imaging systems based on windows and over PC's.

The execution time and the communication time could be lessened by using parallel slackness. Which means that we has to spawn a number of processes on the same processor to keep it beasy when the communication is transmitting or receiving data from the farmer. We can also implement the vision pipe by letting the functions of the vision pipe execute in a pipelined manner.

Chapter 7

General conclusion

We have seen a way to the application of parallel processing to medical imaging through the design of parallel software to the elastic matching and the octree visualization algorithms. Both algorithms are necessary in most of the processes involved in medical imaging diagnosis and compute large amount of 2D or 3D data sets generated by NMR scanners.

We have worked a number of solutions to the mssm algorithm with various parallel programming models and ended with the design and implementation of a parallel solution to the octree visualization algorithm that we have named PVS.

Our work required from us the design of a volume data encoding file format called VDE, a granularity computing technique, a convenient method of subdividing volumes suitable for vision based onTunnels.

We have developed many necessary libraries for the implementation of the solutions over transputer platforms for communication and graphics, designed data structures for volume data elements and developed a vision pipe application, which offers a concise software for encoding, mapping and display.

Finally, a software has been written in Visual C++ to simulate the PVS with four processors. The performance of this parallel solution was a function of the granularity G and the number of processors N. The optimum grain size which gives the best performance for medical applications was found at G=16.Therefore, there is

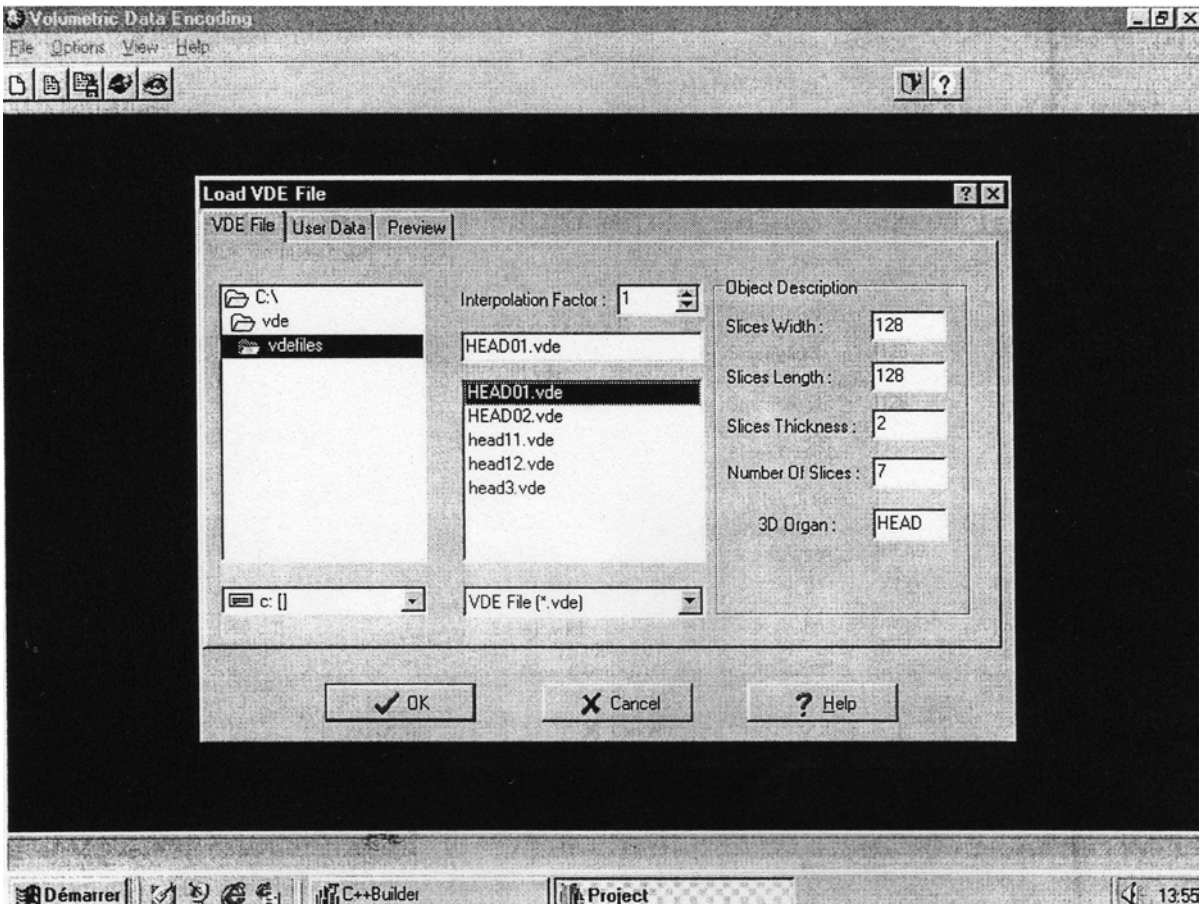
no need to partition beyond that optimum value since the communication overhead will get greater than the running time.

As a conclusion, we have proven that by using adequate partitioning techniques and parallel models we can generalize the application of parallel processing to medical imaging applications over MIMD networks and over LANs of IBM-PCs.

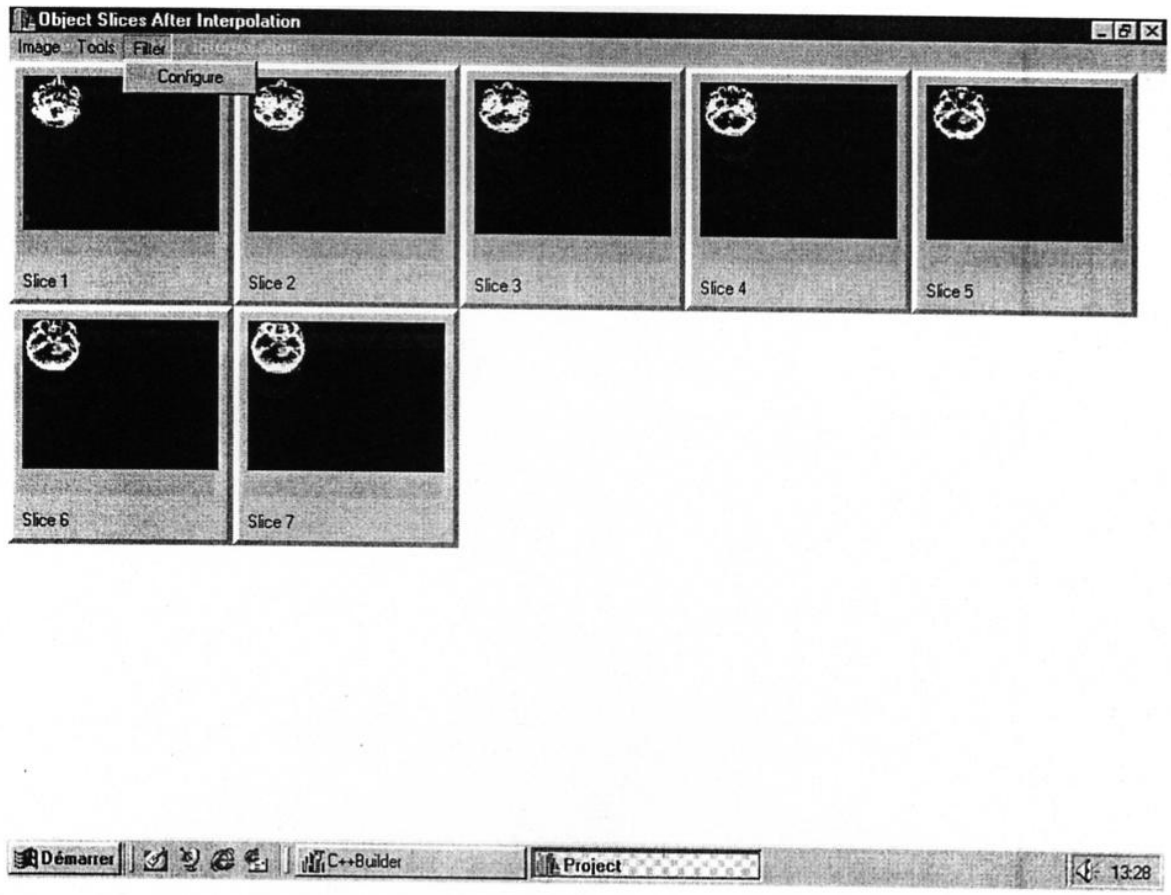
Templates

1. Volumetric data encoding :

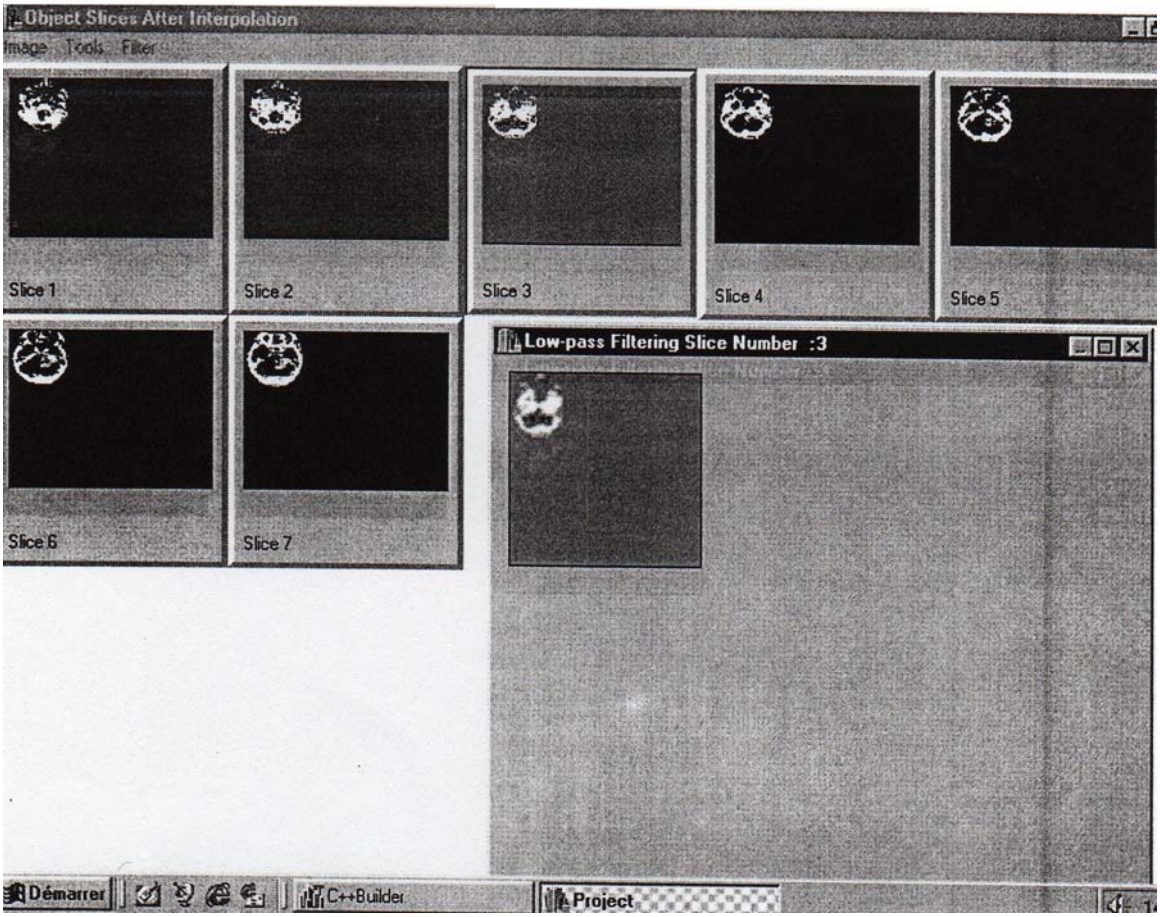
The Volumetric Data Encoding software allows to Load a VDE encoded file such the head1.vde file.



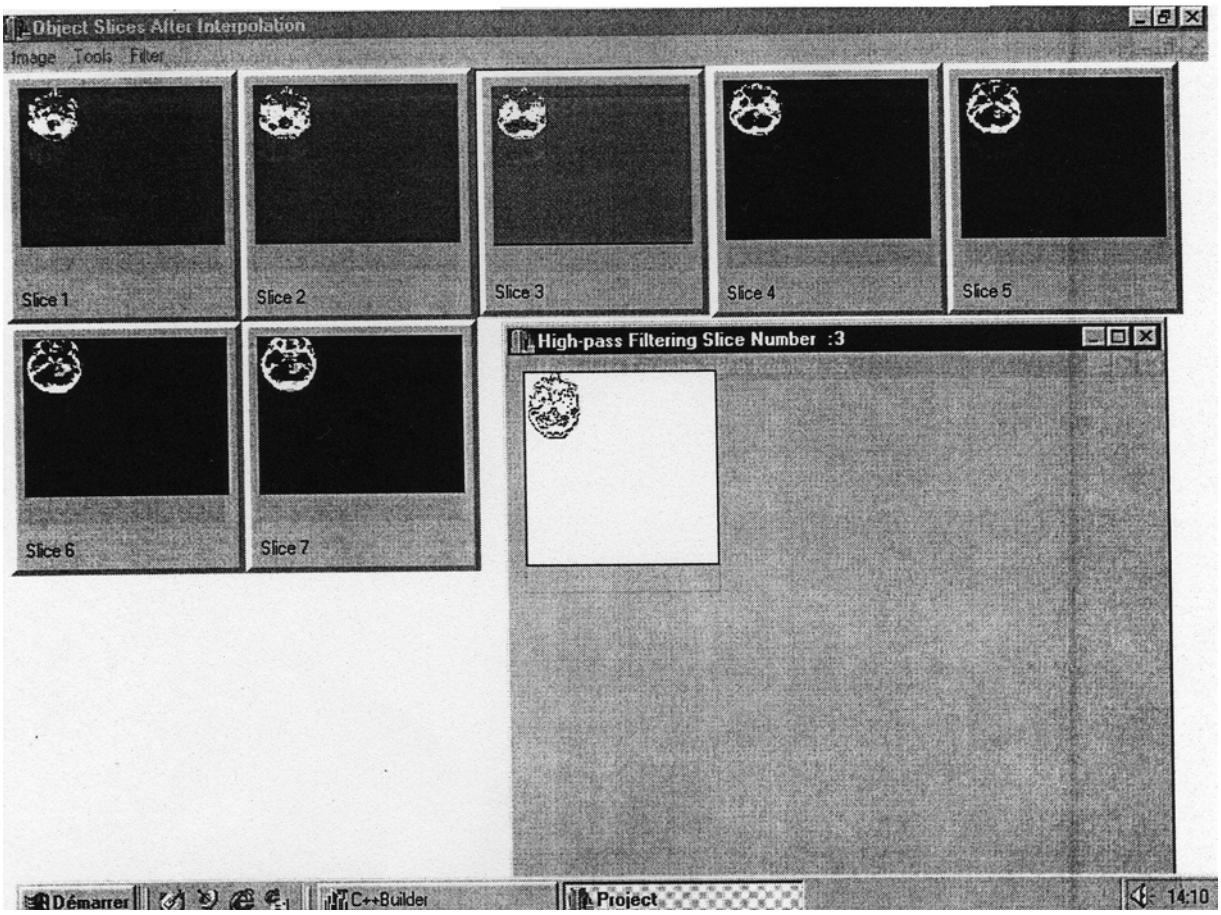
Template 1 : The interface window of the VDE software



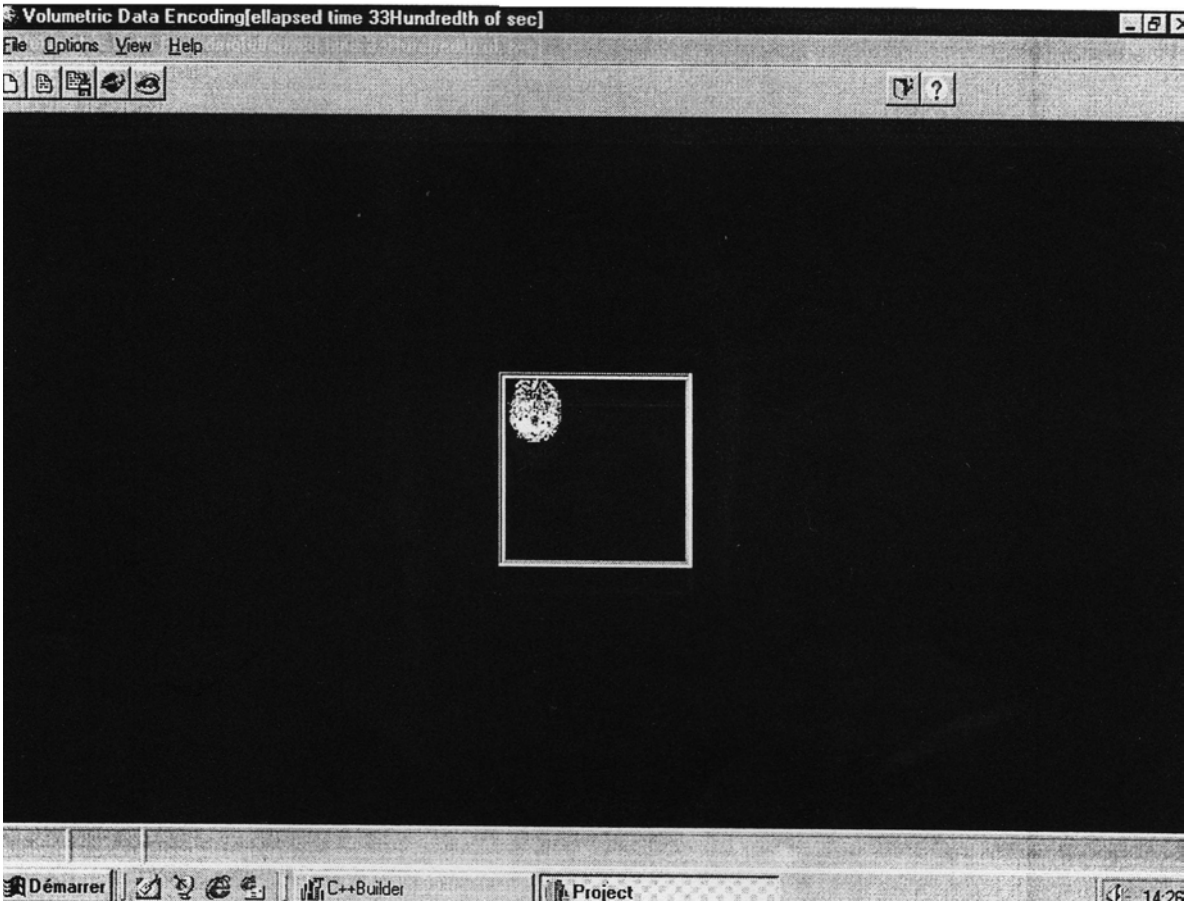
Template 2 : 2D scanned slices of a head [Kennedy87]



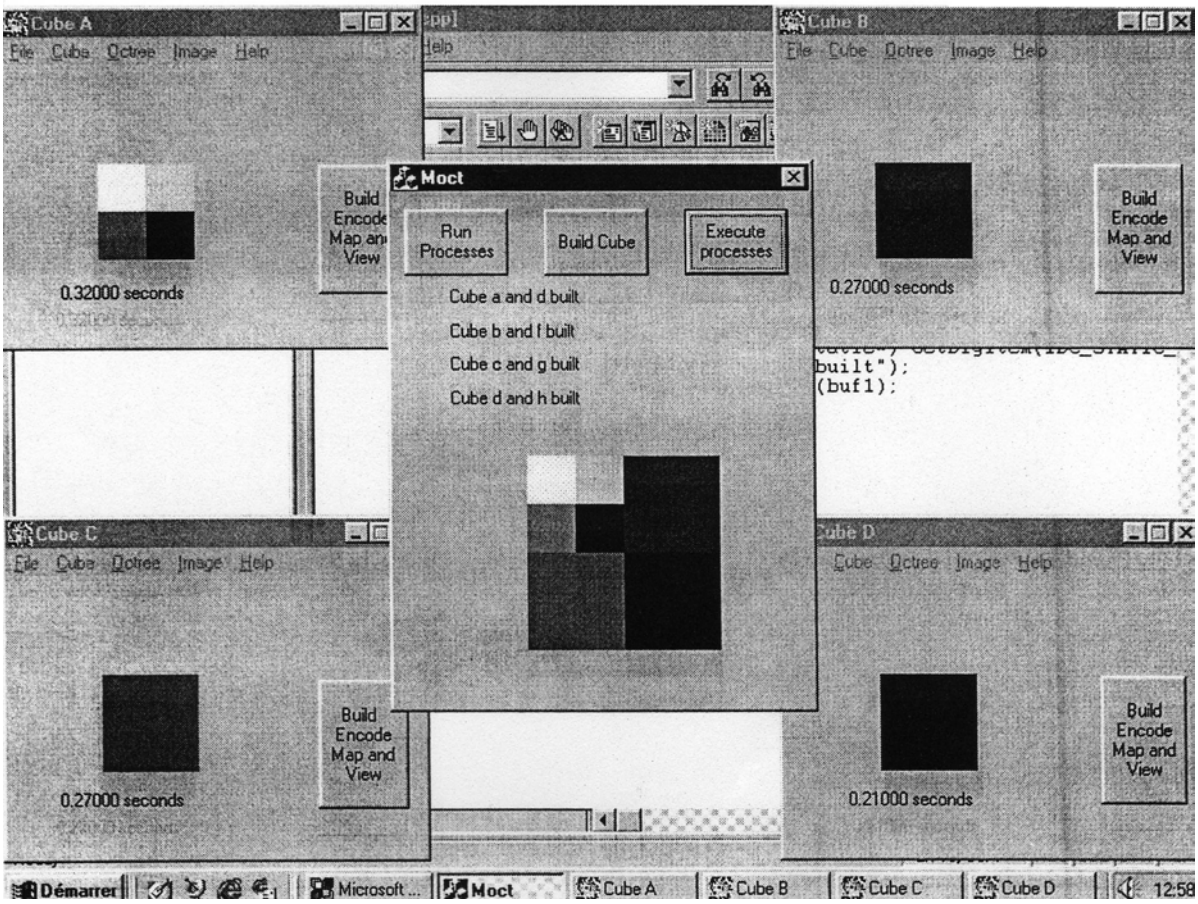
Template 3 : Low pas filtering (slice number 3 of template2)



Template 4 : High-pass filtering (slice number 3 of template 2)



Template 5 : (Visualization of the head.)



Template 6: This template shows the interface window of the Simulation of PVS over windows 95 written in VISUAL C++. The interface window here shows the software running with four processors. With each assigned a sub volume (cube A, B,C, and D) as shown in the four corners of the interface window. At the end, each process sends the visualized image to a process for filling the buffer image (shown in the center of the interface window).

REFERENCES

[3lltd88]
3L Ltd , 1988, ”Parallel C user guide ”, 3L Ltd Peel House, Ladywell, Livingston EH546AG, Scotland, UK.

[Ahuja88]
Ahuja S., Carriero N.J., Gelernter D.H. and Krishnaswany V., August 1988, ”Matching language and hardware for parallel computation in Linda machine”, IEEE transactions on computers, Vol. 37(8).

[Ainsworth91]
Ainsworth Mark, 1991, ”User Manual for the SMT101 Sprint Board: a B004 standard transputer board for IBMPC”, Sundance Multiprocessor Technology Ltd.

[Andrews83]
Andrews G.R.,Schneider F.B., 1983, ”Concepts and notations for concurrent programming”, ACM Computing Surveys, Vol. 15(1), pp. 3-43.

[Andrew90]
Andrew S. Tanenbaum, , ”Structured computer organization”, ISBN : 0-13-852872-1.

[Askew88]
Askew Charlie, Eddition 1988, ”Occam and the transputer - research and applications”, IOS, Springfield, VA, ISBN: 22152-2848.

[Bakkers89]
Bakkers Andre, Edition 1989, ”Applying transputer based parallel machines”, Edited by IOS, Amsterdam, Springfield, VA, QA76.8.A64.1989.

[Bajcsy83]
Bajcsy Ruzena, Lieberson R., Reivich M., August 1983, ”A Computerized System for the Elastic Matching of Deformed Radiographic Images to Idealized Atlas Images”, Journal of Computer Assisted Tomography, Vol. 7(4), pp. 618-625.

[Bajcsy89]
Bajcsy R., Kovacic S., 1989, ”Multiresolution Elastic Matching”, Computer Vision, Graphics, and Image Processing, Vol. 46, pp. 1-21.

[Beaulieu89]
Beaulieu J.M., Goldberg M., 1989, ”Hierarchy in picture segmentation: A stepwise optimization approach”, IEEE Trans. Pattern Anal. Machine Intell., vol. 11(2), pp. 150-163.

[Ben-Ari82]
Ben-Ari M., Edition 1982, ”Principles of concurrent programming”, New Jersy, Prentice Hall International Inc., ISBN: 0-13-711821-X.

[Bisiani88]
Bisiani R., Nowatzyk A., Ravishankar M., December 1988, ”Coherent Shared Memory on a Message Passing Machine”, Technical report CMU-CS-88-204, School of Computer Science, Carnegie Melon University, PA 15213.

[Bershad91]
Bershad Brian N., Zekanskas Matthew J., September 1991, ”Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors”, School of Computer Science, Carnegie Mellon University, CMU-CS-91-170, Pittsburgh, PA 15213.

[Borges84]
Borges A.R.O.S., Ferrari de Almeida A.M.B., 1984, ”Computed tomography: a multi-microprocessor approach”, In Proc. of 1984 Moroccan Workshop on Signal Processing and its Applications, pp. A3/1.1-1.16, Morocco.

[Borges87]
Borges A.R.O.S., Ferrari de Almeida A.M.B., 1987, ”A Multiprocessor architecture for image reconstruction in CT”, Universidade de Aveiro, Parallel Processing for Computer Vision and Display.

[Burkhardt91]
Burkhardt H., Neuvo Y., Simon J.C., Edition 1991, ”From pixels to features II: Parallelism in Image Processing”, Proceedings of a Workshop held at Bonas France, August 27-September 1, 1990, ISBN:0-444-89003-3.

[Burt81]
P.J. Burt, T. Hong, and A. Rosenfeld, 1981, ”Segmentation and estimation of image region properties through cooperative hierarchical computation”, IEEE Trans. Syst., Man, Cybern., vol. SMC-11, no.12,pp.802-809.

[Carriero89]
Carriero N., Gelernter D., April 1989, ”Linda in Context”, Communication of the ACM, Vol. 32(4).

[Chandy88]
Chandy K. Mani, Mistra Jayader, 1988, ”Parallel program design, A foundation”, QA76.6.C4281.

[Charef94]
Charef Sid Ahmed, August 1994, ‘Parallelization of the mssm algorithm using process decomposition’, Master thesis, Inelec, Algeria.

[Chaudhuri89]
Chaudhuri S., et al, Sept 1989,’Detection of blood vessels in retinal images using two-dimentional matched filters’, IEEE trans. on Medical Imaging, vol. 8, No.3.

[Cho93]
Cho Z. H., Jones J. P., Singh M., 1993, ’Foundations of medical imaging’, John Wiley & sons, Inc, ISBN:0-471-54573-2.

[Cline88]
Cline H.E., Lorensen W.E., Ludke S., Crawford C.R., and Teeter B.L., 1988, ”Two algorithms for three dimensional reconstruction of tomograms. Medical Physics vol 15(3) pp. 320-327.

[Crowley ..]
Crowley J.L. , Parker A.C., ..., ” A representation for shape based on peaks and ridges in the difference of low-pass transform,” IEEE Trans. Pattern Anal. Machine Intell., vol. PAMI-6(2), pp. 156-169.

[Culloch88]
Culloch A.D., March 1988, ‘Parallel Programming Toolkit for 3L-C, Fortran and Pascal’, Technical report, 3L Ltd, Peel House, Livingston, EH54 6AG, Scotland, UK.

[CSA90]
Computer System Architects (CSA) , 1990, ”Logical Systems C for transputer, Version 89.1 user manual”.

[Davies91]
Davies Ian, Edition 1991, ”The Helios parallel operating system”, Written by members of the Helios group at Perihelion Software Ltd, Prentice Hall International Inc., ISBN:0-13-381237-5.

[Dev85]
Dev P., Fellingham L.L., Wood S.L., Vassiliadis A., 1985, ”A medical graphics system for diagnosis and surgical planning”, Computer Assisted Radiology 85 (eds. Lemke H.U., Rhodes M.L., Jaffe C.C., Felix R.), Springer-Verlag (Berlin), pp. 602-607.

[Fanibunda83]

Fanibunda K., 1983, ”Photo radiography of Facial Structures”, Brit. Journal of Oral Surgery Vol. 21,pp246-258.

[Ferraro88]
Ferraro Richard F., Edition 1988, ”Programmer’s Guide to the EGA and VGA Cards”, Addison-Wesley Publishing Company Inc., ISBN:0-201-12692-3.

[Fishman89]
Fishman E.K., Magid D.,Ney D.R., Kuhlman J.E., 1989, ”Three-Dimensional imaging: Orthopedic Applications”, In: 3D Imaging in Medicine, (eds:Udupa J.K., Herman G.T.), Lewis Publishers Inc. (Chelsea, MI).

[Flynn83]
Flynn M, Matteson R, Dickie D, Keyes J.W., Bookstein F., 1983, ”Requirements for the Display and Analysis of three-Dimensional Medical Image Data. SPIE Picture Archiving and Communication Systems for Medical Applications, Vol. 418, pp. 213-224.

[Freider85]
Freider G., Gordon D., Reynolds R., 1985, ”Back-To-Front Display of Voxel-Based Objects”, IEEE Computer Graphics and Applications, Vol 5(1), pp. 52-59.

[Fuch88]
Fuchs H., 1988, ”VLSI-intensive Graphics Systems. In: Mathematics and Computer Science in Medical Imaging” (eds: Viergever MA, Todd-Pokropek A) NATO ASI Series F, Springer-Verlag (Berlin), pp. 221-240.

[Gehringer87]
Gehringer E.F., Siewiorek D.P., Segall Z., Edition 1987, ”Parallel processing: The Cm* Experience”, Digital Equipment Corporation, Digital Press, ISBN:1-55558-019-X.

[Gelernter85]
Gelernter David, January 1985, ‘Generative communication in Linda’, ACM transactions on programming languages and systems, Vol. 7(1), pp. 80-112.

[Gerbessiotis94]
Gerbessiotis A.V., Valiant L.G., 1994, ”Direct Bulk-Synchronous Parallel Algorithms”, Journal of Parallel and Distributed Computing Vol. 22, pp. 251-267.

[Gibbons89]
Gibbons P.B., June 1989, ”A More Practical PRAM model”, SPAA 89 Proceedings of the 1989 Symposium on Parallel Algorithms and Architectures, Santa Fe, USA, pp. 158-168.

[Goldwasser87]
Goldwasser S. M. , Reynolds R. A. , 1987, ”Real-Time Display and Manipulation of 3D Medical Objects: The voxel Processor Architecture”, Computer Vision, Graphics, and Image Processing, Vol 39, pp. 1-27.

[Gonzalez87]
Gonzalez Rafael C., Edition 1987, ”Digital Image Processing”, Addison-Wesley Publishing Company Incorporation, ISBN:

[Gorsline87]
Gorsline George W., Edition 1987, ‘Computer Organization: Hardware/Software’, Prentice Hall International, Inc, ISBN:0-13-165796-8.

[Gottlieb83]
Gottlieb Allan, Grishman R., Kruskal Clyde P., McAuliffe Kevin P., Rudolph Larry, and Snir Marc, 1983, ”The NYU Ultra computer-Designing an MIMD Shared Memory Parallel Computer”, IEEE transactions on computers, Vol.C-32(2).

[Green88]
Green S.A., Paddon D.J., September 1988, ”An extension of the Processor Farm Using a Tree Architecture”, Department of Computer Science, University of Bristol, Proceedings of the 9th OCCAM User Group.

[Hearn94]
Hearn D., Baker M. P., ‘ Computer graphics’, Prentice Hall International Ltd., ISBN:0-13-159690-x.

[Harris94]
Harris Tim J., June 1994, ‘ A survey of PRAM Simulation Techniques’, Computer Science Department, University of Edinburgh, EH93JZ, Scotland, ACM Computing surveys, Vol. 26(2), pp.187-206.

[Held96]
Held Gilbert, 1987, ”Data and image compression tools and techniques” , Book, QA76.9.D33.H473.

[Hemmy87]
Hemmy D.C., Lindquist T.R., 1987, ”Optimizing 3-D imaging techniques to meet clinical requirements, NCGA 1987, Conference Proceedings, III, Technical Sessions, pp. 69-80.

[Hillis85]
Hillis W.D., Edition 1985, ”The Connection Machine”, Massachusetts, MIT press, ISBN:0-262-08157-1.

[Hoar85]
Hoar C.A.R. , Edition 1985, ”Communicating Sequential Processes”, United Kingdom, Printice Hall International Ltd, ISBN:0-13-153271-5.

[Hockney88]
Hockney R.W., Jesshope C.R., Edition 1988, ”Parallel computers 2: architecture, programming and algorithms”, Pennsylvania, IOP publishing Ltd, ISBN:0-85274-811-6.

[Hwang84]
Hwang Kai, Briggs Faye A., Edition 1984, ”Computer architecture and parallel processing”, McGraw-Hill series in computer organization and architecture, ISBN:0-07-031556-6.

[Inmos85]
Inmos Ltd, Colin Whitby-Strevens, 1985, ”The transputer”, Inmos Limited, White friars, Lewins Mead, Bristol, BS1 2NP, UK, IEEE 1985.

[Inmos88a]
Inmos Ltd, Edition 1988, ”Occam2 Reference Manual”. Prentice Hall International Series, ISBN:0-13-629312-3.

[Inmos88b]
Inmos Ltd, Edition 1988, ”The transputer data book”, 1st edition, Inmos Ltd, Bristol.

[Inmos88c]
Inmos Ltd, Edition 1988, ”Transputer Development System”, Prentice Hall International, ISBN:0-13-928995-x.

[Inmos90a]
Inmos Ltd, 1990, ”IMS B008 User guide and reference manual ”, Inmos document numbers:72 TRN 223 00, 19990.

[Inmos90b]
Inmos Ltd, January 1990, ‘ S708 User Guide ‘, 72 OEK 227 01.

[Inmos90c]
Inmos Ltd, August 1990, ”ANSI C Toolset User Manual, Reference Manual, Handbook, IMSD7214 IBMPC Manual, Release notes”, Inmos document numbers:72 TDS 22400, 22500, 22600, 22700, 29300.

[Inmos90d]
Inmos Limited, ‘IMS D7214 IBM/NEC PC ANSIC toolset delivery manual’, 72 TDS 22700, August 1990.

[Inmos92]

Inmos Ltd, January 13,19992, ”UBIK Users’ Guide :The open standard for scalable parallel programming”, Inmos document number: IN.WP4.

[Jackel85]
Jackel D., 1985, ”The graphics PARCUM system a 3D memory based computer architecture for processing and display of solid models”, Computer Graphics Forum, Vol 4(1), pp. 21-32.

[Jain89]
Jain A. K., 1989, ‘Fundamentals of digital image processing’, University of California, Davis, Prentice Hall, Englewood Cliffs, NJ 07632, ISBN:0-13-336165-9.

[James90]
James L. Turley, 1990, ”Advanced 80386 programming techniques”, Osborne McGraw-Hill, Berkeley, California, 1988, ISBN : 0-07-8811342-5.

[Jones92]
Jones A., 1992, ”Aspects of Hashing in Address Translation”, Draft Report, Inmos Ltd.

[Kaufman86]
Kaufman, 1986, ‘ A survey of architectures designed for rendering voxel data ‘& 1988, ‘Cubic frame buffer’.

[Kelly89]
Kelly Paul, Edition 1989, ”Fundamental programming for Loosely-Coupled multiprocessors, QA76.6.K449.

[Kennedy87]
Kennedy David N., Nelson C. Alan, June 1987, ”Three-Dimensional Display from Cross-Sectional Tomographics Images: An Application to Magnetic Resonance Imaging”, Medical Imaging, Vol. 6(2).

[Khodja98]
Khodja Mohamed , Bouklahi Abbes, 1998, ‘Visualization of 3D human body’, Final project thesis, Inelec, Algeria.

[Krishnamoorthy94]
Krishnamoorthy S., Choudhary A., 1994, ”A Scalable Distributed Shared Memory Architecture”, Journal of Parallel and Distributed Computing, Vol.22, pp. 547-554.

[Kronsjo86]
Kronsjo L., Edition 1986, ”Computational Complexity of Sequential and Parallel Algorithms”, Wiley , ISBN:471-90814-2.

[Kumar92]
Kumar Vipin, Gupta Anshul, October 30, 1992, ”Analysing Scalability of Parallel Algorithms and Architectures”, Department of Computer Science, University of Minnesota, Minneapolis, MN, 55455, USA.

[Kung88]
Kung H.T., August 28,1987, ”Computational models for parallel computers”, Department of computer science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15123, CMU-CS-88-164.

[Lenz86]
Lenz R., Gudmundsson B., Lindskog B., Danielson P-E, 1986, ”Display of Density Volumes”, IEEE Computer Graphics and Applications, Vol 6(7), pp. 20-29.

[Levoy88]
Levoy M., 1988, ”Volume Rendering - Display of surfaces from Volume data. IEEE Computer Graphics and Applications, Vol 8(3), pp. 29-37.

[Levoy90]
Levoy M, 1990, ‘ Volume Rendering - A hybrid Ray tracer for rendering polygon and volume data’, IEEE computer Graphics & Applications.

[Lifshitz90]
Lifshitz L.M., Pizer S.M., June 1990, ” A multi resolution Hierarchical approach to Image Segmentation Based on Intensity Extrema,”, IEEE Trans. on pattern analysis and machine intelligence, Vol.12(6).

[Lust87]
Lust E., Overbeek, et al, Edition 1987, ”Portable programs for Parallel Processors”, Holt, Rinehart, and Winston, ISBN:0-03-014153-2.

[Mallon91]
Mallon David P., December 12, 1991, ”Communicating through Shared Objects”, School of Computer Studies, University of Leeds, UK.

[Margulis90]
Margulis Neal, March 1990, ”i860 microprocessor internal architecture”, Intel Ltd, In Microprocessors and Microsystems, Vol.14(2).

[Marr80]
Marr D, Hildreth E., 1980, ‘ Theory of edge detection’, Proc. Roy. Soc. London, Ser. B, Vol. 207, pp 187-217.

[Marr82]
Marr David, Edition 1982, ”Vision: A Computational Investigation into the Human Representation and Processing of Visual Information”, San Francisco, CA:W.H. Freeman, QP475.M27.1982.

[May87]
May D., and Shepherd R., 1987, ”Communicating Process Computers”, Inmos Technical Note 22, Inmos Limited, Bristol.

[May89]
May D., October 14, 1989, ”Toward General Purpose Computers”, Inmos Ltd Report.

[May90]
May D. Thompson P., May 9, 1990, ”Transputers and Routers: Components for Concurrent Machines”, Inmos Ltd Report.

[Marsan86]
Marsan M., Baldo G., Conte G., Edition 1986, ”Performance models of multiprocessor systems”, Massachussets, MIT press, ISBN:0-2262-01093-3.

[Meagher82]
Meagher D.J., 1982, ”Geometric Modeling Using Octree Encoding”, Computer Graphics and Image Processing, Vol. 19, pp. 129-147.

[Meagher84a]
Meagher D.J., 1984, ”Interactive Solids Processing for Medical Analysis and Planning”, Proceedings of the National Computer Graphics Association, Vol. II, pp. 96-106.

[Meagher84b]
Meagher D.J., 1984, ”The Solids Engine: A Processor for Interactive Solid Modelling”, Proceedings of the Nicograph ‘84 Tokyo, November.

[Meagher85]
Meagher D.J., 1985, ”INSIGHT: Special purpose hardware system for 3D medical graphics”, Rennsalear Polytechnic Albany.

[Meiko89a]
Meiko Ltd, Vardas, January 26,1989, ”A programmer’s introduction to SUN-CSTOOLS”, Draft, Bristol, UK, Meiko Ltd.

[Meiko89b]
Meiko Limited, 1989, ”A tutorial introduction to CSTOOLS”, Report, S0205-06S.03, Meiko Ltd.

[Mills90]

Mills James G., October 22, 1990, ”A guide to using Bodyscan”, Edinburgh parallel computing center, University of Edinburgh.

[Morris88]
Morris D.T., 1988, ”Parallel Algorithms and Architectures for the display of constructive solid geometry”, PhD. Dissertation , University of Leeds, Dept. of Computer Science.

[Mougari99]
Mougari A., Laroui A., Bouklachi Abbes, ‘ Volumetric data encoding’, final year project report, department of electronic and electrical engineering, University of Boumerdes.

[Mowforth89]
Mowforth P.H., Zhengping A.J., September 1989, ”Model Based Tissue Differentiation in MR Brain Images”, The Turring Institute 36 North Hanover Street, Glasgow, G12AD, Proceeding of the 5th alvey vision conference, Vol.25-28 pp.67-71.

[Murray90]
Murray William D., Edition 1990, ”Computer and digital system architecture”, QA76.9.A73.M981.1990.

[Nash91]
Nash J.M., Dew P.M., July 12, 1991, ”XPRAM Model and Programming interface”, Technical report, School of Computer Studies, The University of Leeds, UK.

[Parberry87]
Parberry Ian., Edition 1987, ”Parallel Complexity theory”, London, John Wiley & Sons, ISBN:0-470-20937-3.

[Perrott87]
Perrott R.H., Edition 1987, ”Parallel programming ”, Massachusetts, Addison - Wesley, ISBN:0-201-14231-7.

[Pizer86]
Pizer S.M., et al, 1986, ” An image description for object definition, based on extremal regions in the stack,’ in Proc. 9th Conf. Information Processing in Medical Imaging. Boston, MA:Martinus Nijhoff.

[Pountain90]
Pountain Dick, April 1990, ”Virtual Channels: The next generation of transputers”, BYTE.

[Quinn87]
Quinn Michael J., Edition 1987, ”Designing efficient algorithms for parallel computers”, McGraw Hill international editions, Computer science series. QA76.5.Q56.1987.

[Parker83]
Parker J.A., Kenyon R.V., and Troxel D.E., March 1983, ”Comparison of interpolating methods for image resampling”, IEEE transactions on medical imaging, vol. MI-2(1).

[Reynolds87]
Reynolds R.A., Gordon D., Chen L.S., 1987, ”A Dynamic Screen Technique for Shaded Graphics Display of Solid-Represented Objects”, Computer Vision, Graphics, and Image Processing, Vol. 38, pp. 275-298.

[Rhodes87]
Rhodes M.L., Kuo Y.M., Rothman S.L.G., Woznick C., 1987, ”An application of computer graphics and networks to anatomic model and prosthesis manufacturing”, IEEE Computer Graphics and Applications, vol. 7(2), pp. 12-25.

[Robb87]
Robb R. A., 1987, ”A workstation for interactive display and analysis of multidimensional biomedical images”, Computer Assisted Radiology 87 (eds. Lemke H.U., Rhodes M.L., Jaffe C.C., Felix R.) Springer-Verlag (Berlin), pp. 642-657.

[Robb89]

Robb R.A., Barollot C., September 1989, ”ANALYSE: Interactive display and analysis of 3D medical images”, IEEE transaction on medical imaging, Vol. 8(3).

[Salmon87]
Salmon R, et al, 1987, ‘ Computer graphics system and concepts’, Addison-Wesley Inc.

[Samet88a]
Samet H., Webber R.E., 1988, ”Hierarchical Data Structures and Algorithms for Computer Graphics, Part 1: Fundamentals”, IEEE Computer Graphics and Applications, Vol. 8(7), pp. 48-68.

[Samet88b]
Samet H. Webber R.E., 1988, ”Hierarchical Data Structures and Algorithms for Computer Graphics, Part 2: Applications”, IEEE Computer Graphics and Applications, Vol 8(7), pp. 59-75.

[Seitz85]
Seitz Charles L., January 1985, ”The Cosmic Cube”, Communications of the ACM, Vol.28(1).

[Srihari81]
Srihari S. N., 1981, ”Representation of three-Dimensional Digital Images”, Computing Surveys, Vol. 13(4), pp. 399-424.

[Stone77]
Stone H.S., 1977, ”Multiprocessor scheduling with the aid of network flow”, IEEE transactions, Software Eng. Vol.SE-3, pp.85-93.

[Stone87]
Stone Harold S., Edition 1987, ”High performance computer architecture”, Massachusetts, Adison - Wesley Publishing Company, ISBN:0-21-16802-2.

[Swan92]
Swan Tom, Edition 1992, ”Mastering Borland C++, Version 3.1”, Prentice Hall Computer Publishing, ISBN:0-672-30274-8.

[Tisher97]
Tisher Michael, Jennrich Bruno,1997, ‘ The PC bible system programming’ by Micro application Inc., 6th edition.

[Udupa81]
Udupa J.K.,1981, ”Determination of 3D Shape Parameters from Boundary Information”, Computer Graphics and Image processing , Vol.17, pp. 52-59.

[Udupa82]
Udupa J.K.,1982, ”Interactive Segmentation and Boundary Surface Formation for 3-D Digital Images”, Computer Graphics and Image processing, Vol.18, pp. 213-235.

[Udupa86]
Udupa J.K., Herman G.T., Chen L.S., Margasahayem P.S., Myer C.R., 1986, ”3D98: a turnkey system for the 3D display and analysis of medical objects in CT data, Proc. SPIE, Vol. 671, pp.154-168.

[Udupa87]
Udupa J.K., 1987, ”3D Imaging in Medicine”, Proceedings of NCGA Computer Graphics 87, 9th annual conference and exposition, Philadelphia, Tutorials Volume II, pp. 73-105.

[Valiant88]
Valiant L.G., January 8,1988, ”General Purpose Parallel Architectures”, TR-07-89, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, USA.

[Valiant89]
Valiant Leslie G., April 13, 1989, ”Bulk Synchronous Parallel Computers”, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138, USA.

[Valiant90]

Valiant L.G., August 1990, ”A Bridging Model for Parallel Computation’, Communications of the ACM, Vol. 33(8).

[Vannier83]
Vannier M.W. , Marsh J.L., Warren J.O., 1983, ”Three-Dimensional Computer Graphics for Crano-Facial Surgical Planning and Evaluation”, Computer Graphics, Vol. 17(3), pp. 263-272.

[Wedd87]
Wedd S., Sutcliffe J., Burkinshaw L., Horsman A., March ,1987, ”Tomographic Reconstruction from Experimentally Obtained Cone-Beam”, IEEE transaction on medical imaging, Vol. MI-6(1).

[Williams90]
Williams Shirley A., Edition 1990, ”Programming models for parallel systems”, Wiley, ISBN:745-80338-5.

[Yves93]
Yves Meyer, Edition 1993, ”Wavelets Algorithms & Applications”, Translated and revised by Robert D. Ryan, SIAM, ISBN:0-89871-309-9.

[Zair92]
Zair Abdelouahab, Mallon D.P., Dew P.M., 1992, ”Actel: A concurrent object based language”, School of Computer Studies, The University of Leeds, In Proceedings of the ECOOP Workshop on dynamic object placement and load balancing.