Practical Parallel Processing for Realistic Rendering

Tim Davis Alan Chalmers Henrik Wann Jensen

Course #30 SIGGRAPH 2000

New Orleans, USA
23-28 July 2000

About The Authors

Timothy Davis is an assistant professor in the Computer Science Department at Clemson University where he works within the newly established MFAC (Master of Fine Arts in Computing) group, which trains students to produce special effects for entertainment and commercial projects. Most recently, he has published papers at the 1999 IEEE Parallel Visualization and Graphics Symposium and the 1998 Eurographics Workshop on Parallel Graphics and Visualisation. His research involves exploiting spatio-temporal coherence in a parallel environment to reduce rendering times for ray-traced animations. He received his Ph.D. from North Carolina State University and has worked in technical positions for the Environmental Protection Agency and NASA Goddard Space Flight Center.

Alan Chalmers is a senior lecturer in the Department of Computer Science at the University of Bristol, UK. He has published over 70 papers in journals and international conferences on parallel photo-realistic graphics. He was the co-chairman of the recent IEEE Parallel Visualization and Graphics Symposium and is chairman of the Eurographics workshop series on Parallel Graphics and Visualisation.Recently he and Professor F. W. Jansen were the guest editors of the Journal of Parallel Computing for its special edition on Parallel Graphics and Visualisation. He is also currently vice-chair of ACM SIGGRAPH. His research interests include the application of parallel photo-realistic graphics to archaeological site visualisation in order to provide a flexible tool for investigating site reconstruction and utilization.

Henrik Wann Jensen is a research associate in the computer graphics laboratory at Stanford where he is working with Pat Hanrahan on global illumination techniques for complex environments and parallel rendering algorithms. Prior to that he was a postdoc in the computer graphics group at MIT. He received his M.Sc. in 1993 and his Ph.D. in 1996 at the Technical University of Denmark for developing the photon map algorithm.

Contents

Acknowledgements

We are very grateful to International Thomson Computer Press for permission to use material from:

Chalmers A.G., Tidmus J.P. *Practical Parallel Processing: An Introduction to Problem Solving in Parallel*. International Thomson Computer Press, 327 pages, ISBN 1-85032-135-3, March 1996.

Chapter 1

Structure of the Course

Parallel processing offers the potential of rendering high-quality images and animations in reasonable times. This course begins by reviewing the basic issues involved in rendering within a parallel or distributed computing environment. Specifically, various methods are presented for dividing the original rendering problem into subtasks and distributing them efficiently to independent processors. Careful consideration must be taken to balance the processing load across the processors, as well as reduce communication between these subtasks for faster processing. The course continues by examining the strengths and weaknesses of multiprocessor machines and networked render farms for graphics rendering. Case studies of working applications will be presented to demonstrate in detail practical ways of dealing with the issues involved.

Introduction (15 minutes) (Davis)

- The need for speed in satisfying the demand for high-quality graphics
- Rendering and parallel processing: a holy union
- The necessity of parallel processing in a world of advanced 3D graphics cards

Parallel/Distributed Rendering Issues (30 minutes) (Chalmers)

- Task subdivision
- Load balancing
- Communication
- Task migration
- Data Management

Classification of Parallel Rendering Systems

By task subdivision: (30 minutes) (Davis)

- Polygon rendering
  - Sort first
  - Sort middle
  - Sort last
- Ray tracing
  - Image space subdivision

  - Object space subdivision
  - Object subdivision

By hardware: (30 minutes) (Jensen)

- Parallel Hardware
  - Machines currently in use
  - Pros and cons in using parallel hardware for rendering
- Distributed Computing
  - Using commodity workstations and PCs for high-performance rendering Render farms
  - Pros and cons in using distributed systems for rendering

Practical Applications

- Parallel radiosity and ray tracing (30 minutes) (Chalmers)
- Distributed computing and spatial/temporal coherence (30 minutes) (Davis)
- Simple and efficient global illumination using photon maps (30 minutes) (Jensen)

Summary, discussion and questions (variable) (All)

1.1 Structure of the notes

The notes contain important background information as well as detailed descriptions of the Parallel Processing techniques described. The notes are arranged as follows:

Chapter 2 introduces the concepts which are necessary to understand the difficulties confronting any parallel implementation.

Chapter 3 describes the issues associated with scheduling tasks on a parallel system. A number of ways of decomposing a problem on a parallel machine are considered and the advantages and disadvantages of each approach are highlighted.

Chapter 4 concentrates on managing large data requirements which are distributed across the parallel environment. Issues of consistency and latency are considered.

Chapter 5 considers the classification of parallel rendering systems according to the method of task subdivision and/or by the hardware used.

Finally, some detailed notes are presented on “Global Illumination in Complex Scenes using Photon Maps”, including a discussion on the parallel implementation of this technique.

Chapter 2

Introduction

Parallel processing is like a dog's walking on its hind legs. It is not done well, but you are surprised to find it done at all.

[Steve Fiddes (University of Bristol) with apologies to Samuel Johnson]

Realistic computer graphics is an area of research which develops algorithms and methods to render images of artificial models or worlds as realistically as possible. Such algorithms are known for their unpredictable data accesses and their high computational complexity. Rendering a single high quality image may take several hours, or even days. Parallel processing offers the potential for solving such complex problems in reasonable times.

However, there are a number of fundamental issues: task scheduling, data management and caching techniques, which must be addressed if parallel processing is to achieve the desired performance when computing realistic images. These are applicable for all three rendering techniques presented in this tutorial: ray tracing, radiosity and particle tracing.

This chapter introduces the concepts of parallel processing, describes its development and considers the difficulties associated with solving problems in parallel.

Parallel processing is an integral part of everyday life. The concept is so ingrained in our existence that we benefit from it without realising. When faced with a taxing problem, we involve others to solve it more easily. This co-operation of more than one worker to facilitate the solution of a particular problem may be termed parallel processing. The goal of parallel processing is thus to solve a given problem more rapidly, or to enable the solution of a problem that would otherwise be impracticable by a single worker.

The principles of parallel processing are, however, not new, as evidence suggests that the computational devices used over 2000 years ago by the Greeks recognised and exploited such concepts. In the Nineteenth Century, Babbage used parallel processing in order to improve the performance of his Analytical Engine [48]. Indeed, the first general purpose electronic digital computer, the ENIAC, was conceived as a highly parallel and decentralised machine with twenty-five independent computing units, co-operating towards the solution of a single problem [27].

However, the early computer developers rapidly identified two obstacles restricting the widespread acceptance of parallel machines: the complexity of construction; and, the seemingly high programming effort required [10]. As a result of these early set-backs, the developmental thrust shifted to computers with a single computing unit, to the detriment

of parallel designs. Additionally, the availability of sequential machines resulted in the development of algorithms and techniques optimised for these particular architectures.

The evolution of serial computers may be finally reaching its zenith due to the limitations imposed on the design by its physical implementation and inherent bottlenecks [5]. As users continue to demand improved performance, computer designers have been looking increasingly at parallel approaches to overcome these limitations. All modern computer architectures incorporate a degree of parallelism. Improved hardware design and manufacture coupled with a growing understanding of how to tackle the difficulties of parallel programming has re-established parallel processing at the forefront of computer technology.

2.1 Concepts

Parallel processing is the solution of a single problem by dividing it into a number of sub-problems, each of which may be solved by a separate worker. Co-operation will always be necessary between workers during problem solution, even if this is a simple agreement on the division of labour. These ideas can be illustrated by a simple analogy of tackling the problem of emptying a swimming pool using buckets. This job may be sub-divided into the repeated *task* of removing one bucket of water.

A single person will complete all the tasks, and complete the job, in a certain time. This process may be speeded-up by utilising additional workers. Ideally, two people should be able to empty the pool in half the time. Extending this argument, a large number of workers should be able to complete the job in a small fraction of the original time. However, practically there are physical limitations preventing this hypothetical situation.

The physical realisation of this solution necessitates a basic level of co-operation between workers. This manifests itself due to the contention for access to the pool, and the need to avoid collision. The time required to achieve this co-operation involves inter-worker communication which detracts from the overall solution time, and as such may be termed an *overhead*.

2.1.1 Dependencies

Another factor preventing an ideal parallel solution are termed: dependencies. Consider the problem of constructing a house. In simple terms, building the roof can only commence after the walls have been completed. Similarly, the walls can only be erected once the foundations are laid. The roof is thus dependent upon the walls, which are in turn dependent on the foundations. These dependencies divide the whole problem into a number of distinct stages. The parallel solution of each stage must be completed before the subsequent stage can start.

The dependencies within a problem may be so severe that it is not amenable to parallel processing. A strictly sequential problem consists of a number of stages, each comprising a single task, and each dependent upon the previous stage. For example, in figure 2.2, building a tower of toy blocks requires a strictly sequential order of task completion. The situation is the antithesis of dependency-free problems, such as placing blocks in a row on the floor. In this case, the order of task completion is unimportant, but the need for co-operation will still exist.

Pipelining is the classic methodology for minimising the effects of dependencies. This technique can only be exploited when a process, consisting of a number of distinct stages,

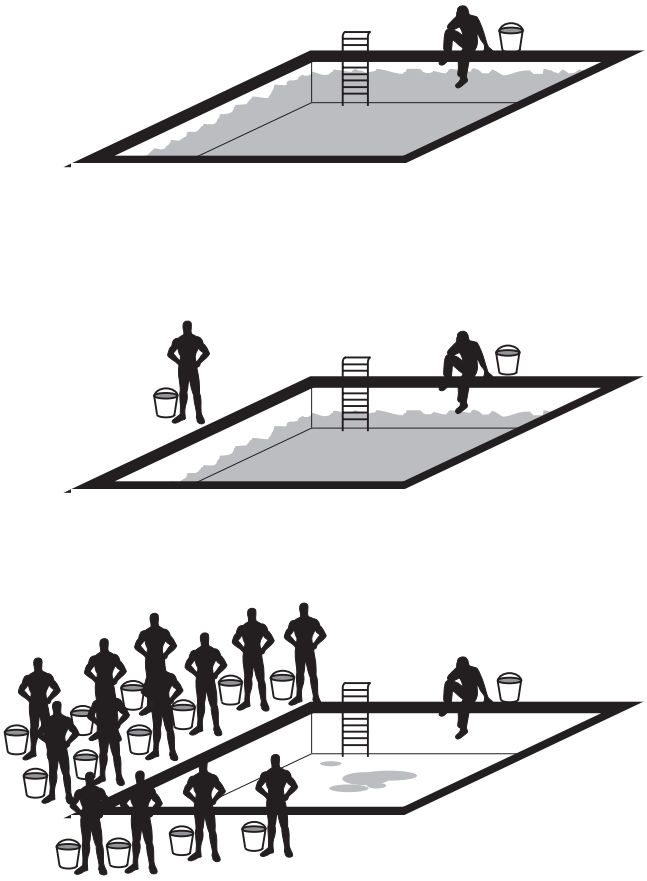


Figure 2.1: Emptying a pool by means of a bucket

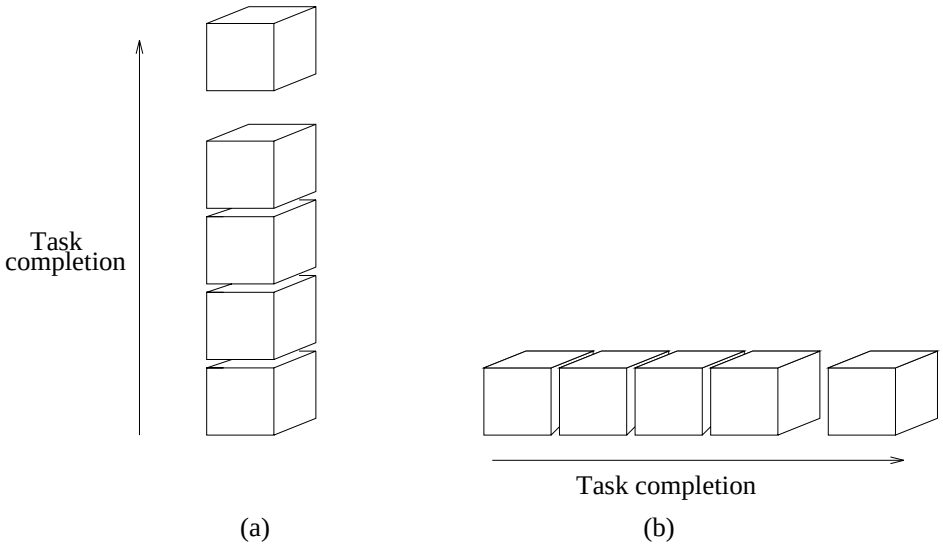


Figure 2.2: Building with blocks: (a) Strictly sequential (b) dependency-free

needs to be repeated several times. An automotive assembly line is an example of an efficient pipeline. In a simplistic form, the construction of a car may consist of four linearly dependent stages as shown in figure 2.3: chassis fabrication; body assembly; wheel fitting; and, windscreen installation. An initial lump of metal is introduced into the pipeline then, as the partially completed car passes each stage, a new section is added until finally the finished car is available outside the factory.

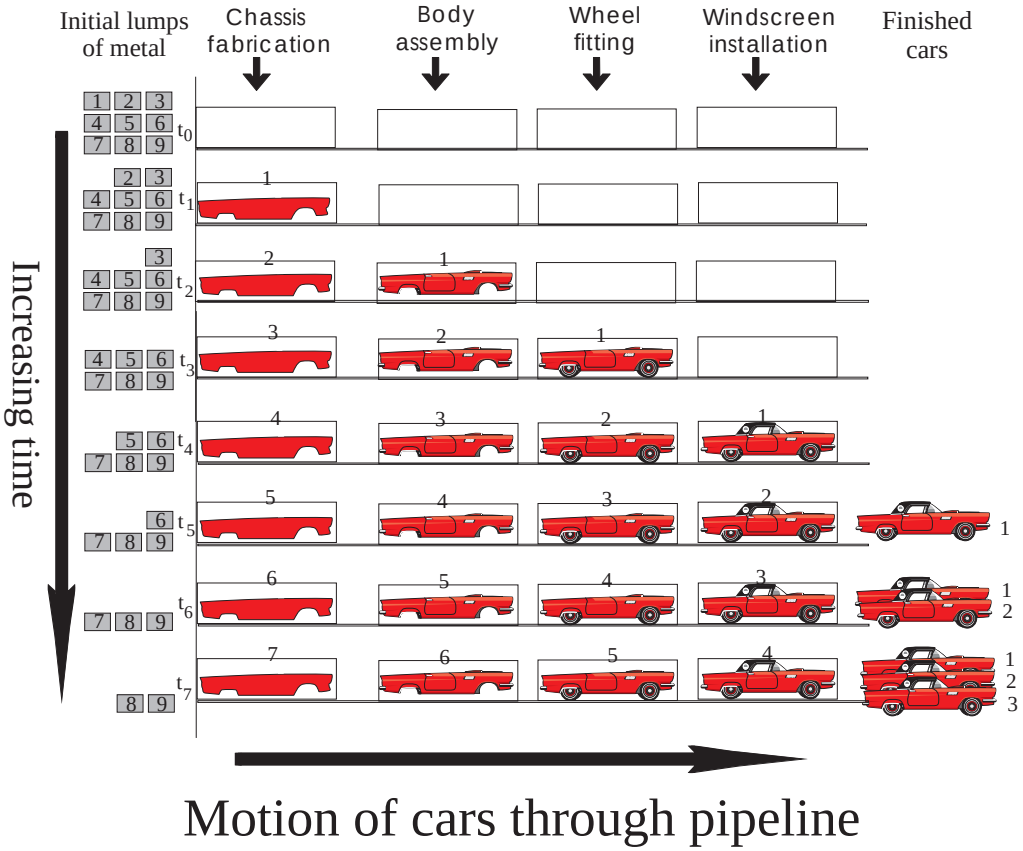


Figure 2.3: Pipeline assembly of a car

Consider an implementation of this process consisting of four workers, each performing their task in one time unit. Having completed the task, the worker passes the partially completed car on to the next stage. This worker is now free to repeat its task on a new component fed from the previous stage. The completion of the first car occurs after four time units, but each subsequent car is completed every time unit.

The completion of a car is, of course, sensitive to the time taken by each worker. If one worker were to take longer than one time unit to complete its task then the worker after this difficult task would stand idle awaiting the next component, whilst those before the worker with the difficult task would be unable to move their component on to the next stage of the pipeline. The other workers would thus also be unable to do any further work until the difficult task was completed. Should there be any interruption in the input to the pipeline then the pipeline would once more have to be “refilled” before it could operate at maximum efficiency.

2.1.2 Scalability

Every problem contains a upper bound on the number of workers which can be meaningfully employed in its solution. Additional workers beyond this number will not improve solution time, and can indeed be detrimental. This upper bound provides an idea as to how suitable a problem is to parallel implementation: a measure of its *scalability*.

A given problem may only be divided into a finite number of sub-problems, corresponding to the smallest tasks. The availability of more workers than there are tasks, will not improve solution time. The problem of clearing a room of 100 chairs may be divided into 100 tasks consisting of removing a single chair. A maximum of 100 workers can be allocated one of these tasks and hence perform useful work.

The optimum solution time for clearing the room may not in fact occur when employing 100 workers due to certain aspects of the problem limiting effective worker utilisation. This phenomenon can be illustrated by adding a constraint to the problem, in the form of a single doorway providing egress from the room. A *bottleneck* will occur as large numbers of workers attempt to move their chairs through the door simultaneously, as shown in figure 2.4.

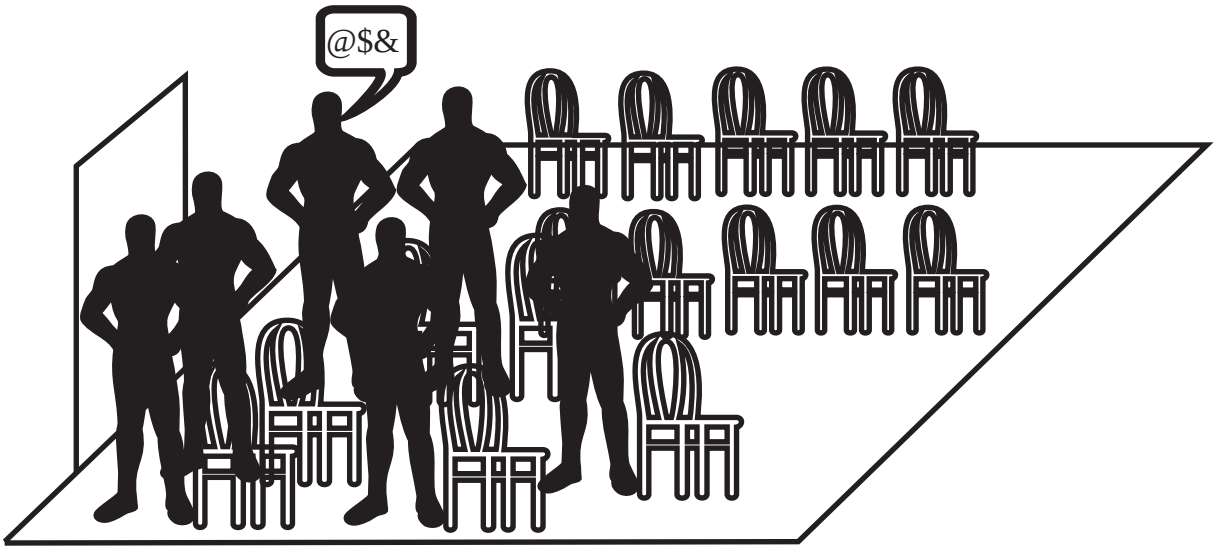


Figure 2.4: Bottleneck caused by doorway

The delays caused by this bottleneck may be so great that the time taken to empty the room of chairs by this large number of workers may in fact be *longer* than the original time taken by the single worker. In this case, reducing the number of workers can alleviate the bottleneck and thus reduce solution time.

2.1.3 Control

All parallel solutions of a problem require some form of control. This may be as simple as the control needed to determine what will constitute a task and to ascertain when the problem has been solved satisfactorily. More complex problems may require control at several stages of their solution. For example, solution time could be improved when clearing the room if a controller was placed at the door to schedule its usage. This control

would ensure that no time was wasted by two (or more) workers attempting to exit simultaneously and then having to "reverse" to allow a single worker through. An alternative to this explicit *centralised* control would be some form of *distributed* control. Here the workers themselves could have a way of preventing simultaneous access, for example, if two (or more) workers reach the door at the same time then the biggest worker will always go first while the others wait.

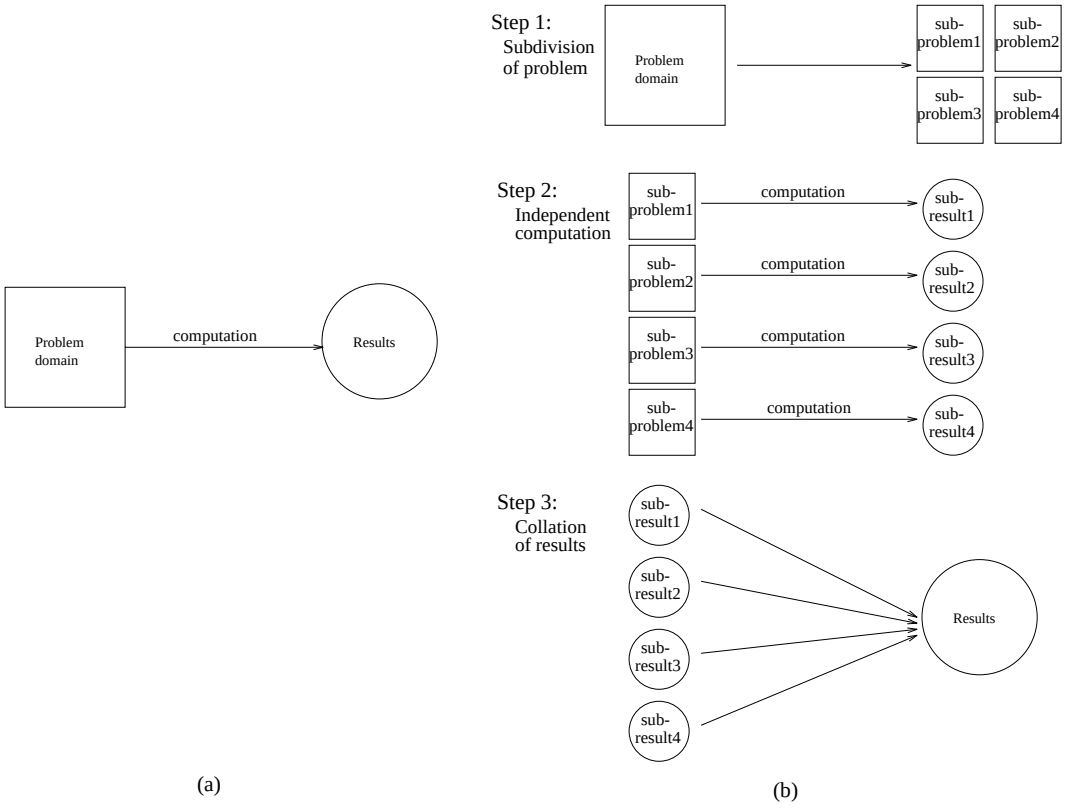


Figure 2.5: Control required in (a) a Sequential versus (b) a parallel implementation

Figure 2.5(a) shows the sequential approach to solving a problem. Computation is applied to the problem domain to produce the desired results. The controlled parallel approach shown in figure 2.5(b) achieves a parallel implementation of the same problem via three steps. In step 1, the problem domain is divided into a number of sub-problems, in this case four. Parallel processing is introduced in step 2 to enable each of the sub-problems to be computed in parallel to produce sub-results. In step 3, these results must now be collated to achieve the desired final results. Control is necessary in steps 1 and 3 to divide the problem amongst the workers and then to collect and collate the results that the workers have independently produced.

2.2 Classification of Parallel Systems

A traditional sequential computer conforms to the *von Neumann* model. Shown in figure 2.6, this model comprises a processor, an associated memory, an input/output interface and various busses connecting these devices. The processor in the von Neumann model

is the single computational unit responsible for the functions of fetching, decoding and executing a program's instructions. Parallel processing may be added to this architecture through pipelining using multiple functional units within a single computational unit or by replicating entire computational units (which may contain pipelining). With pipelining, each functional unit repeatedly performs the same operation on data which is received from the preceding functional unit. So in the simplest case, a pipeline for a computational unit could consist of three functional units, one to fetch the instructions from memory, one to decode these instructions and one to execute the decoded instructions. As we saw with the automobile assemblage example, a pipeline is only as effective as its slowest component. Any delay in the pipeline has repercussions for the whole system.

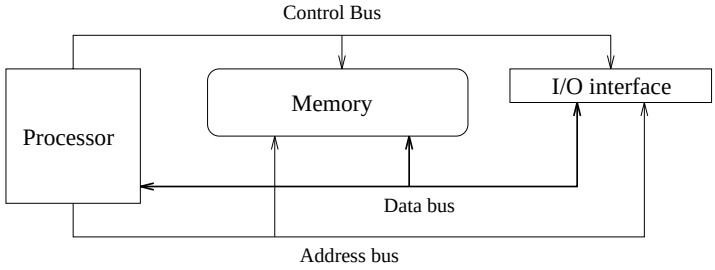


Figure 2.6: Von Neumann model architecture

Vector processing was introduced to provide efficient execution of program loops on large array data structures. By providing multiple registers as special vector registers to be used alongside the central processing unit, a vector processor is able to perform the *same operation* on *all elements* of a vector simultaneously. This simultaneous execution on every element of large arrays can produce significant performance improvements over conventional *scalar processing*. However, often problems need to be reformulate to benefit from this form of parallelism. A large number of scientific problems, such as weather forecasting, nuclear research and seismic data analysis, are well suited to vector processing.

Replication of the entire computational unit, the *processor*, allows individual tasks to be executed on different processors. Tasks are thus sometimes referred to as *virtual processors* which are allocated a physical processor on which to run. The completion of each task contributes to the solution of the problem.

Tasks which are executing on distinct processors at any point in time are said to be running in *parallel*. It may also be possible to execute several tasks on a single processor. Over a period of time the impression is given that they are running in parallel, when in fact at any point in time only *one* task has control of the processor. In this case we say that the tasks are being performed *concurrently*, that is their execution is being shared by the same processor. The difference between parallel tasks and concurrent tasks is shown in figure 2.7.

The workers which perform the computational work and co-operate to facilitate the solution of a problem on a parallel computer are known as *processing elements* and are often abbreviated as *PEs*. A processing element consists of a processor, one or more tasks, and the software to enable the co-operation with other processing elements. A parallel system comprises of *more than one* processing element.

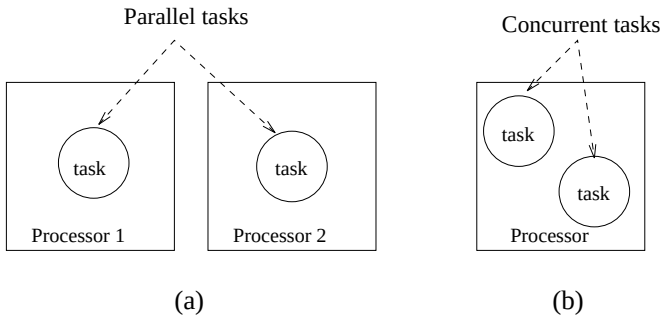


Figure 2.7: (a) Parallel tasks (b) Concurrent tasks

2.2.1 Flynn's taxonomy

The wide diversity of computer architectures that have been proposed, and in a large number of cases realised, since the 1940's has led to the desire to classify the designs to facilitate evaluation and comparison. Classification requires a means of identifying distinctive architectural or behavioural features of a machine.

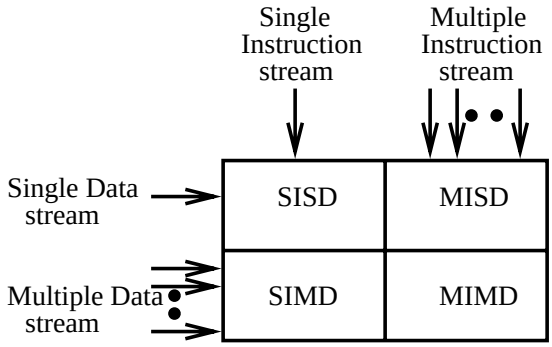


Figure 2.8: Flynn's taxonomy for processors

In 1972 Flynn proposed a classification of processors according to a macroscopic view of their principal interaction patterns relating to instruction and data *streams* [21]. The term stream was used by Flynn to refer to the sequence of instructions to be executed, or data to be operated on, by the processor. What has become known as *Flynn's taxonomy* thus categorises architectures into the four areas shown in figure 2.8.

Since its inception, Flynn's taxonomy has been criticised as being too broad and has thus been enlarged by several other authors, for example, Shore in 1973 [55], Treleaven, Brownbridge and Hopkins in 1982 [58], Basu in 1984 [7], and perhaps one of the most detailed classifications was given by Hockney and Jesshope in 1988 [31].

Real architectures are, of course, much more complex than Flynn suggested. For example, an architecture may exhibit properties from more than one of his classes. However, if we are not too worried about the minute details of any individual machine then Flynn's taxonomy serves to separate fundamentally different architectures into four broad categories. The classification scheme is simple (which is one of the main reasons for its popularity) and thus useful to show an overview of the concepts of multiprocessor com-

puters.

SISD: Single Instruction Single Data embraces the conventional sequential, or von Neumann, processor. The single processing element executes instructions sequentially on a single data stream. The operations are thus ordered in time and may be easily traced from start to finish. Modern adaptations of this uniprocessor use some form of pipelining technique to improve performance and, as demonstrated by the Cray supercomputers, minimise the length of the component interconnections to reduce signal propagation times [54].

SIMD: Single Instruction Multiple Data machines apply a single instruction to a group of data items simultaneously. A master instruction is thus acting over a vector of related operands. A number of processors, therefore, obey the same instruction in the same cycle and may be said to be executing in strict *lock-step*. Facilities exist to exclude particular processors from participating in a given instruction cycle. Vector processors, for example the Cyber 205, Fujitsu FACOM VP-200 and NEC SX1, and array processors, such as the DAP [53], Goodyear MPP (Massively Parallel Processor) [8], or the Connection Machine CM-1 [30], may be grouped in this category.

MISD: Multiple Instruction Single Data Although part of Flynn's taxonomy, no architecture falls obviously into the MISD category. One the closest architecture to this concept is a pipelined computer. Another is systolic array architectures which derives their from the medical term "systole" used to describe the rhythmic contraction of chambers of the heart. Data arrives from different directions at regular intervals to be combined at the "cells" of the array. The Intel iWarp system was designed to support systolic computation [4]. Systolic arrays are well suited to specially designed algorithms rather than general purpose computing [40, 41].

MIMD: Multiple Instruction Multiple Data The processors within the MIMD classification autonomously obey their own instruction sequence and apply these instructions to their own data. The processors are, therefore, no longer bound to the synchronous method of the SIMD processors and may choose to operate *asynchronously*. By providing these processors with the ability to communicate with each other, they may interact and therefore, co-operate in the solution of a single problem. This interaction has led to MIMD systems sometimes being classified as *tightly coupled* if the degree of interaction is high, or *loosely coupled* if the degree of interaction is low.

Two methods are available to facilitate this interprocessor communication. *Shared memory* systems allow the processors to communicate by reading and writing to a common address space. Controls are necessary to prevent processors updating the same portion of the shared memory simultaneously. Examples of such shared memory systems are the Sequent Balance [57] and the Alliant FX/8 [17].

In *distributed memory* systems, on the other hand, processors address only their private memory and communicate by passing messages along some form of communication path. Examples of MIMD processors from which such distributed memory systems can be built are the Intel i860 [50],the Inmos transputer [33] and Analog Devices SHARC processor.

The conceptual difference between shared memory and distributed memory systems of MIMD processors is shown in figure 2.9. The interconnection method for the

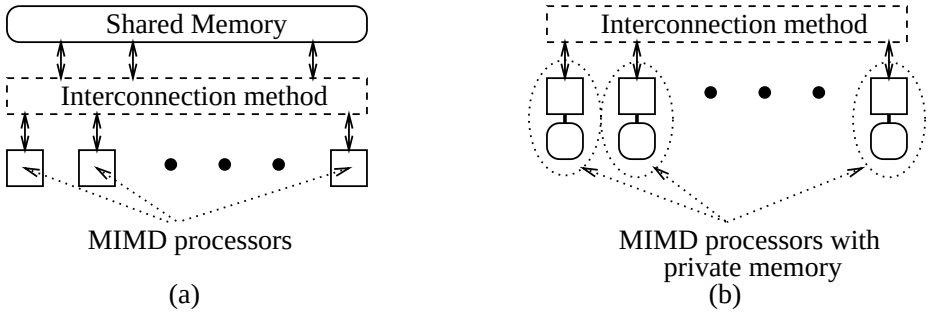


Figure 2.9: Systems of MIMD processors (a) shared memory (b) distributed memory

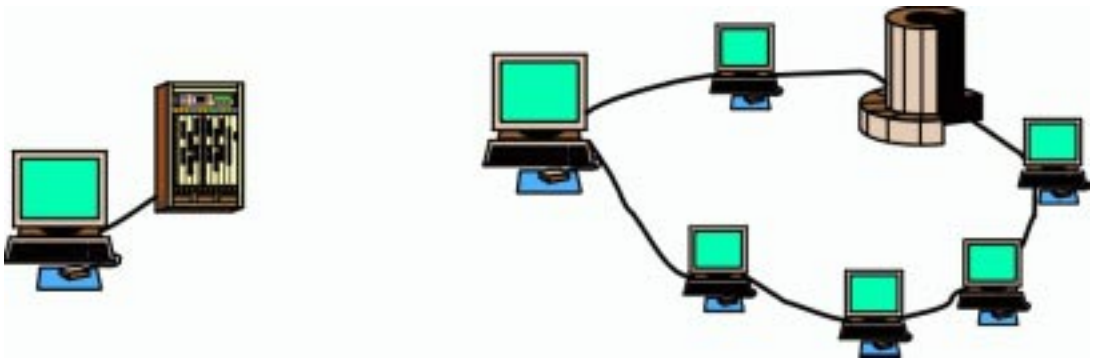
shared memory system, figure 2.9(a), allows all the processors to be connected to the shared memory. If two, or more, processors wish to access the same portion of this shared memory at the same time then some arbitration mechanism must be used to ensure only one processor accesses that memory portion at a time. This problem of memory contention may restrict the number of processors that can be interconnected using the shared memory model. The interconnection method of the distributed memory system, figure 2.9(b), connects the processors in some fashion and if one, or more, processors wish to access another processor's private memory, it, or they, can only do so by sending a message to the appropriate processor along this interconnection network. There is thus no memory contention as such. However, the density of the messages that result in distributed memory systems may still limit the number of processors that may be interconnected, although this number is generally larger than that of the shared memory systems.

Busses have been used successfully as an interconnection structure to connect low numbers of processors together. However, if more than one processor wishes to send a message on the bus at the same time, an arbiter must decide which message gets access to the bus first. As the number of processors increases, so the contention for use of the bus grows. Thus a bus is inappropriate for large multiprocessor systems. An alternative to the bus is to connect processors via dedicated links to form large networks. This removes the bus-contention problem by spreading the communication load across many independent links.

2.2.2 Parallel versus Distributed systems

Distributed memory MIMD systems consist of autonomous processors together with their own memory which co-operate in the solution of a single complex problem. Such systems may consist of a number of interconnected, dedicated processor and memory nodes, or interconnected "stand-alone" workstations. To distinguish between these two, the former configuration is refered to as a (dedicated) *parallel* system, while the latter is known as a *distibuted* system, as shown in figure 2.10.

The main distinguishing features of these two systems are typically the computation-to-communication ratio and the cost. Parallel systems make use of fast, "purpose-built" (and thus expensive) communication infrastructures, while distributed systems rely on exisiting network facilities such as ethernet, which are significantly slower and suceptible to other non-related traffic.



(a) (b)

Figure 2.10: (a) Parallel system (b) Distributed system

The advantage of distributed systems is that they may consist of a cluster of existing workstations which can be used by many (sequential) users when not employed in a parallel capacity. A number of valuable tools have been developed to enable these workstations to act in parallel, such as Parallel Virtual Machine (PVM), and Message Passing Interface (MPI). These provide an easy framework for coupling heterogeneous computers including, workstations, mainframes and even parallel systems.

However, while some of the properties of a distributed computing system may be different from those of a parallel system, many of the underlying concepts are equivalent. For example, both systems achieve co-operation between computational units by passing messages, and each computational unit has its own distinct memory. Thus, the ideas presented in this tutorial should prove equally useful to the reader faced with implementing his or her realistic rendering problem on either system.

2.3 The Relationship of Tasks and Data

The implementation of any problem on a computer comprises two components:

- the algorithm chosen to solve the problem; and,
- the domain of the problem which encompasses all the data requirements for that problem.

The algorithm interacts with the domain to produce the result for the problem, as shown diagrammatically in figure 2.11.

A sequential implementation of the problem means that the entire algorithm and domain reside on a single processor. To achieve a parallel implementation it is necessary to divide the problem's components in some manner amongst the parallel processors. Now no longer resident on a single processor, the components will have to interact within the multiprocessor system in order to obtain the same result. This co-operation requirement introduces a number of novel difficulties into any parallel implementation which are not present in the sequential version of the same problem.

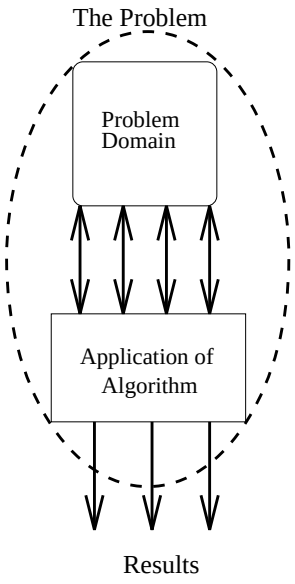


Figure 2.11: The components of a problem

2.3.1 Inherent difficulties

User confidence in any computer implementation of a problem is bolstered by the successful termination of the computation and the fact that the results meet design specifications. The reliability of modern computer architectures and languages is such that any failure of a sequential implementation to complete successfully will point automatically to deficiencies in either the algorithm used or data supplied to the program. In addition to these possibilities of failure, a parallel implementation may also be affected by a number of other factors which arise from the manner of the implementation:

Deadlock: An active parallel processor is said to be deadlocked if it is waiting indefinitely for an event which will never occur. A simple example of deadlock is when two processors, using synchronised communication, attempt to send a message to each other at the same time. Each process will then wait for the other process to perform the corresponding input operation which will never occur.

Data consistency: In a parallel implementation, the problem's data may be distributed across several processors. Care has to be taken to ensure:

- if multiple copies of the same data item exists then the value of this item is kept consistent;
- mutual exclusion is maintained to avoid several processors accessing a shared resource simultaneously; and,
- the data items are fetched from remote locations efficiently in order to avoid processor idle time.

While there is meaningful computation to be performed, a sequential computer is able to devote 100% of its time for this purpose. In a parallel system it may happen that some

of the processors become idle, not because there is no more work to be done, but because current circumstances prevent those processors being able to perform any computation.

Parallel processing introduces communication overheads. The effect of these overheads is to introduce latency into the multiprocessor system. Unless some way is found to minimise communication delays, the percentage of time that a processor can spend on useful computation may be significantly affected. So, as well as the factors affecting the successful termination of the parallel implementation, one of the fundamental considerations also facing parallel programmers is the *computation to communication* ratio.

2.3.2 Tasks

Subdividing a single problem amongst many processors introduces the notion of a task. In its most general sense, a task is a unit of computation which is assigned to a processor within the parallel system. In any parallel implementation a decision has to be taken as to what exactly constitutes a task. The *task granularity* of a problem is a measure of the amount of computational effort associated with any task. The choice of granularity has a direct bearing on the computation to communication ratio. Selection of too large a granularity may prevent the solution of the problem on a large parallel system, while too fine a granularity may result in significant processor idle time while the system attempts to keep processors supplied with fresh tasks.

On completion of a sequential implementation of a problem, any statistics that may have been gathered during the course of the computation, may now be displayed in a straightforward manner. Furthermore, the computer is in a state ready to commence the next sequential program. In a multiprocessor system, the statistics would have been gathered at each processor, so after the solution of the problem the programmer is still faced with the task of collecting and collating these statistics. To ensure that the multiprocessor system is in the correct state for the next parallel program, the programmer must also ensure that all the processors have *terminated gracefully*.

2.3.3 Data

The problem domains of many rendering applications are very large. The size of these domains are typically far more than can be accommodated within the local memory of any processing element (or indeed in the memory of many sequential computers). Yet it is precisely these complex problems that we wish to solve using parallel processing.

Consider a multiprocessor system consisting of sixty-four processing elements each with 4 MBytes of local memory. If we were to insist that the entire problem domain were to reside at each processing element then we would be restricted to solving problems with a maximum domain of 4 MBytes. The total memory within the system is $64 \times 4 = 256$ MBytes. So, if we were to consider the memory of the multiprocessor system as a whole, then we could contemplate solving problems with domains of up to 256 MBytes in size; a far more attractive proposition. (If the problem domain was even larger than this, then we could also consider the secondary storage devices as part of the combined memory and that should be sufficient for most problems.)

There is a price to pay in treating the combined memory as a single unit. Data management strategies will be necessary to translate between the conceptual single memory unit and the physical distributed implementation. The aims of these strategies will be to keep track of the data items so that an item will always be available at a processing element

when required by the task being performed. The distributed nature of the data items will thus be invisible to the application processes performing the computation. However, any delay between the application process requesting an item and this request being satisfied will result in idle time. As we will see, it is the responsibility of data management to avoid this idle time.

2.4 Evaluating Parallel Implementations

The chief reason for opting for a parallel implementation should be: *to obtain answers faster*. The time that the parallel implementation takes to compute results is perhaps the most natural way of determining the benefits of the approach that has been taken. If the parallel solution takes *longer* than any sequential implementation then the decision to use parallel processing needs to be re-examined. Other measurements, such as speed-up and efficiency, may also provide useful insight on the maximum scalability of the implementation.

Of course, there are many issues that need to be considered when comparing parallel and sequential implementations of the same problem, for example:

- Was the same processor used in each case?
- If not, what is the price of the sequential machine compared with that of the multi-processor system?
- Was the algorithm chosen already optimised for sequential use, that is, did the data dependencies present preclude an efficient parallel implementation?

2.4.1 Realisation Penalties

If we assume that the same processor was used in both the sequential and parallel implementation, then we should expect,, that the time to solve the problem decreases as more processing elements are added. The best we can reasonably hope for is that two processing elements will solve the problem twice as quickly, three processing elements three times faster, and n processing elements, n times faster. If n is sufficiently large then by this process, we should expect our large scale parallel implementation to produce the answer in a tiny fraction of the sequential computation, as shown by the "optimum time" curve in the graph in figure 2.12.

However, in reality we are unlikely to achieve these optimised times as the number of processors is increased. A more realistic scenario is that shown by the curve "actual times" in figure 2.12. This curve shows an initial decrease in time taken to solve the example problem on the parallel system up to a certain number of processing elements. Beyond this point, adding more processors actually leads to an *increase* in computation time.

Failure to achieve the optimum solution time means that the parallel solution has suffered some form of *realisation* penalty. A realisation penalty can arise from two sources:

- an *algorithmic* penalty; and,
- an *implementation* penalty.

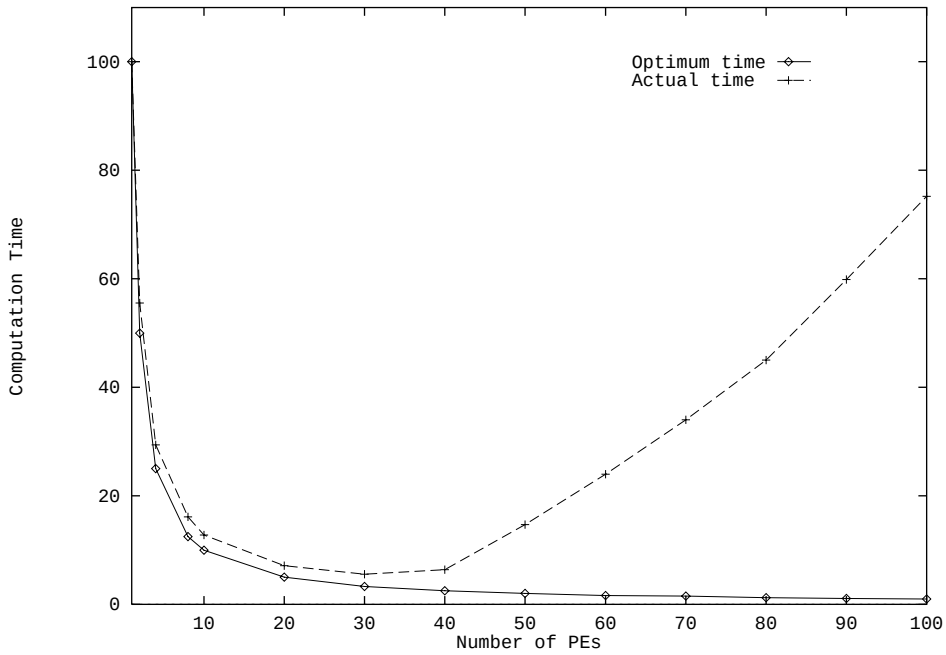


Figure 2.12: Optimum and actual parallel implementation times

The algorithmic penalty stems from the very nature of the algorithm selected for parallel processing. The more inherently sequential the algorithm, the less likely the algorithm will be a good candidate for parallel processing.

Aside: It has also been shown, albeit not conclusively, that the more experience the writer of the parallel algorithm has in sequential algorithms, the less parallelism that algorithm is likely to exhibit [13].

This sequential nature of an algorithm and its implicit data dependencies will translate, in the domain decomposition approach, to a requirement to *synchronise* the processing elements at certain points in the algorithm. This can result in processing elements standing idle awaiting messages from other processing elements. A further algorithmic penalty may also come about from the need to reconstruct sequentially the results generated by the individual processors into an overall result for the computation.

Solving the same problem twice as fast on two processing elements implies that those two processing elements must spend 100% of their time on computation. We know that a parallel implementation requires some form of communication. The time a processing element is forced to spend on communication will naturally impinge on the time a processor has for computation. Any time that a processor cannot spend doing useful computation is an implementation penalty. Implementation penalties are thus caused by:

- **the need to communicate**

  As mentioned above, in a multiprocessor system, processing elements need to communicate. This communication may not only be that which is necessary for a processing element's own actions, but in some architectures, a processing element may also have to act as a intermediate for other processing elements' communication.

- **idle time**

  Idle time is any period of time when an application process is available to perform some useful computation, but is unable to do so because either there is no work locally available, or its current task is suspended awaiting a synchronisation signal, or a data item which has yet to arrive.

  It is the job of the local task manager to ensure that an application process is kept supplied with work. The computation to communication ratio within the system will determine how much time a task manager has to fetch a task before the current one is completed. A *load imbalance* is said to exist if some processing elements still have tasks to complete, while the others do not.

  While synchronisation points are introduced by the algorithm, the management of data items for a processing element is the job for the local data manager. The domain decomposition approach means that the problem domain is divided amongst the processing elements in some fashion. If an application process requires a data item that is not available locally, then this must be fetched from some other processing element within the system. If the processing element is unable to perform other useful computation while this fetch is being performed, for example by means of multi-threading as discussed in section 4.5.2, then the processing element is said to be idle

- **concurrent communication, data management and task management activity**

  Implementing each of a processing element's activities as a separate concurrent process on the same processor, means that the physical processor has to be shared. When another process other than the application process is scheduled then the processing element is not performing useful computation even though its current activity is necessary for the parallel implementation.

The fundamental goal of the system software is to minimise the implementation penalty. While this penalty can never be removed, intelligent communication, data management and task scheduling strategies can avoid idle time and significantly reduce the impact of the need to communicate.

2.4.2 Performance Metrics

Solution time provides a simple way of evaluating a parallel implementation. However, if we wish to investigate the relative merits of our implementation then further insight can be gained by additional metrics. A range of metrics will allow us to compare aspects of different implementations and perhaps provide clues as to how overall system performance may be improved.

Speed-up

A useful measure of any multiprocessor implementation of a problem is *speed-up*. This relates the time taken to solve the problem on a single processor machine to the time taken to solve the same problem using the parallel implementation. We will define the speed-up of a multiprocessor system in terms of the elapsed time that is taken to complete a given problem, as follows:

$$\text{Speed-up} = \frac{\text{elapsed time of a uniprocessor}}{\text{elapsed time of the multiprocessors}} \tag{2.1}$$

The term *linear speed-up* is used when the solution time on an n processor system is n times faster than the solution time on the uniprocessor. This linear speed-up is thus equivalent to the optimum time shown in section 2.4.1. The optimum and actual computation times in figure 2.12 are represented as a graph of linear and actual speed-ups in figure 2.13. Note that the actual speed-up curve increases until a certain point and then subsequently decreases. Beyond this point we say that the parallel implementation has suffered a *speed-down*.

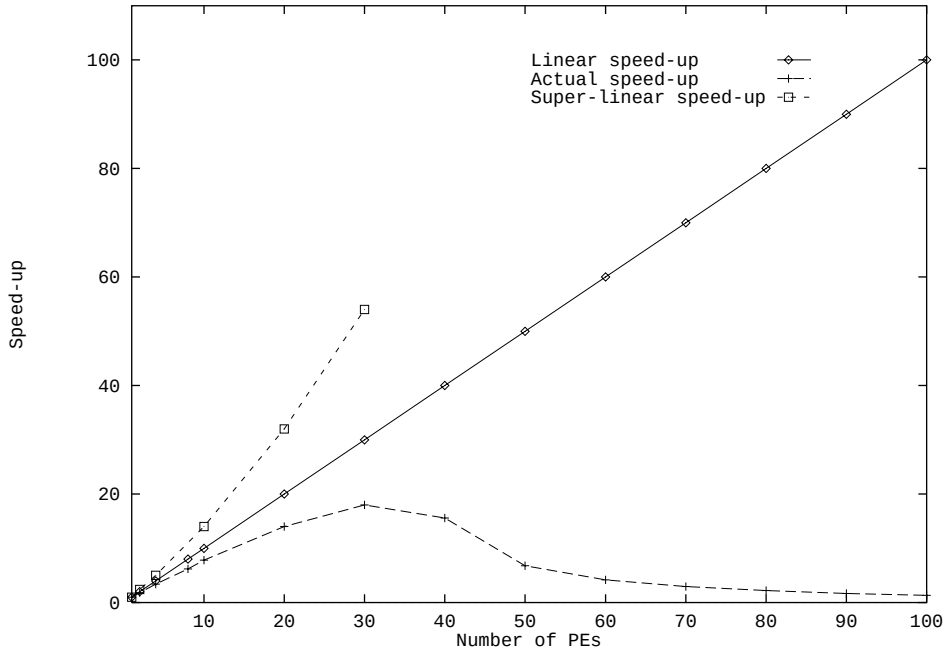


Figure 2.13: Linear and actual speed-ups

The third curve in figure 2.13 represents so-called *super-linear speed-up*. In this example, the implementation on 20 processors has achieved a computation time which is approximately 32 times faster than the uniprocessor solution. It has been argued, see [19], that it is not possible to achieve a speed-up greater than the number of processors used. While in practice it certainly is possible to achieve super-linear speed-up, such implementation may have exploited "unfair" circumstances to obtain such timings. For example, most modern processors have a limited amount of cache memory with an access time significantly faster compared with a standard memory access. Two processors would have double the amount of this cache memory. Given we are investigating a fixed size problem, this means that a larger proportion of the problem domain is in the cache in the parallel implementation than in the sequential implementation. It is not unreasonable, therefore, to imagine a situation where the two processor solution time is more than twice as fast than the uniprocessor time.

Although super-linear speed-up is desirable, in this tutorial we will assume a "fair" comparison between uniprocessor and multiprocessor implementations. The results that

are presented in the case studies thus make no attempt to exploit any hardware advantages offered by the increasing number of processors. This will enable the performance improvements offered by the proposed system software extensions to be highlighted without being masked by any variations in underlying hardware. In practice, of course, it would be foolish to ignore these benefits and readers are encouraged to "squeeze every last ounce of performance" out of their parallel implementation.

Two possibilities exist for determining the "elapsed time of a uniprocessor". This could be the time obtained when executing:

1. an optimised sequential algorithm on a single processor, T_s; or,
2. the parallel implementation on *one* processing element, T_1.

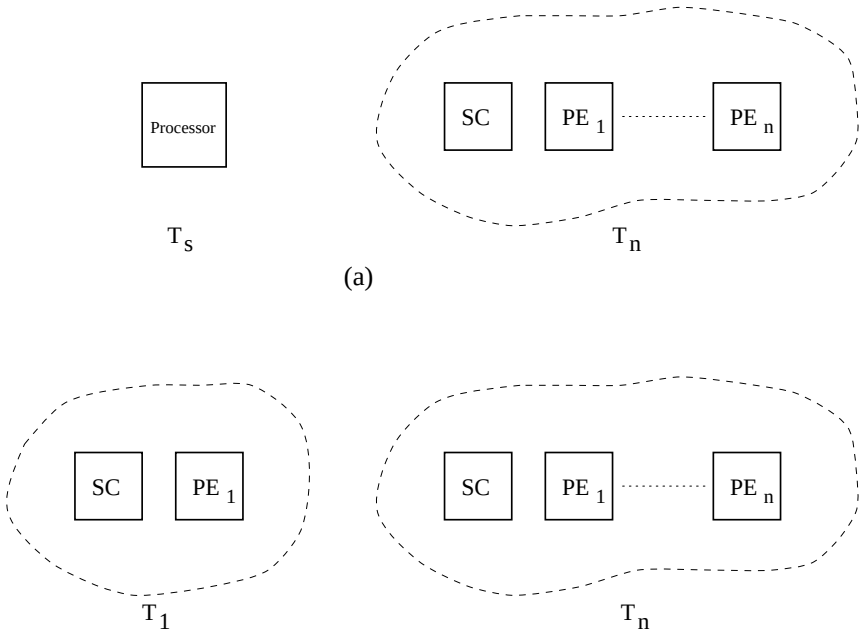


Figure 2.14: Systems used to obtain T_n and (a) T_s (b) T_1

The time taken to solve the problem on n processing elements we will term T_n. The difference between how the two sequential times are obtained is shown in figure 2.14. There are advantages in acquiring both these sequential times. Comparing the parallel to the optimised sequential implementation highlights any algorithmic efficiencies that had to be sacrificed to achieve the parallel version. In addition, none of the parallel implementation penalties are hidden by this comparison and thus the speed-up is not exaggerated. One of these penalties is the time taken simply to supply the data to the processing element and collect the results.

The comparison of the single processing element with the multiple processing element implementation shows how well the problem is "coping" with an increasing number of processing elements. Speed-up calculated as $\frac{T_1}{T_n}$, therefore, provides the indication as to the *scalability* of the parallel implementation. Unless otherwise stated, we will use this alternative for speed-up in the case studies in this book as it better emphasizes the performance improvements brought about by the system software we shall be introducing.

As we can see from the curve for "actual speed-up" in figure 2.13, the speed-up obtained for that problem increased to a maximum value and then subsequently decreased as more processing elements were added. In 1967 Amdahl presented what has become known as "Amdahl's law" [3]. This "law" attempts to give a maximum bound for speed-up from the nature of the algorithm chosen for the parallel implementation. We are given an algorithm in which the proportion of time that needs to be spent on the purely sequential parts is s, and the proportion of time that might be done in parallel is p, by definition. The total time for the algorithm on a single processor is $s + p = 1$ (where the 1 is for algebraic simplicity), and the maximum speed-up that can be achieved on n processors is:

$$\begin{aligned} \text{maximum speed-up} &= \frac{(s+p)}{s+\frac{p}{n}} \\ &= \frac{1}{s+\frac{p}{n}} \end{aligned} \tag{2.2}$$

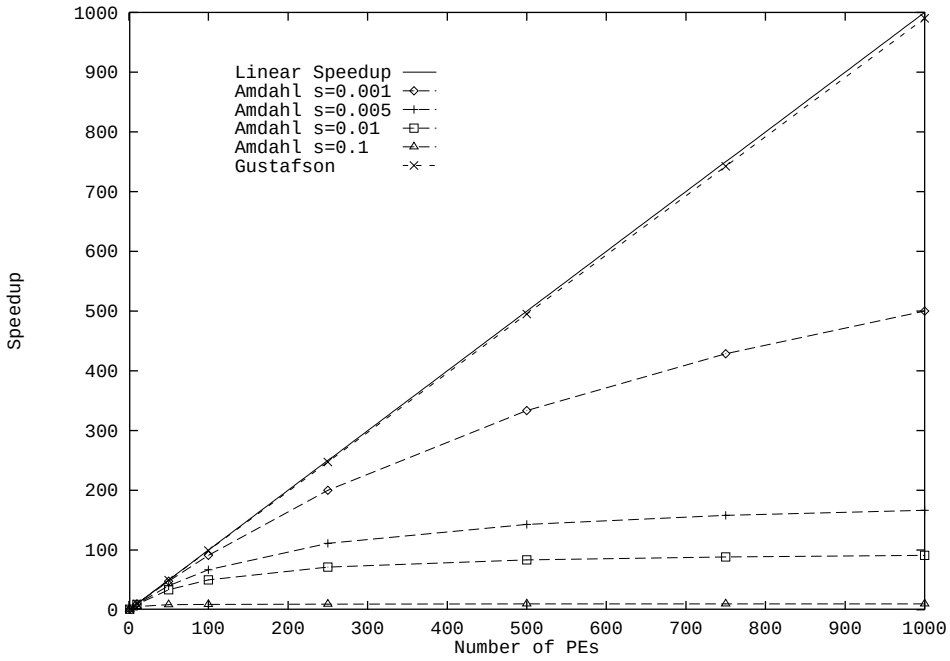


Figure 2.15: Example maximum speed-up from Amdahl and Gustafson's laws

Figure 2.15 shows the maximum speed-up predicted by Amdahl's law for a sequential portion of an algorithm requiring 0.1%, 0.5%, 1% and 10% of the total algorithm time, that is $s = 0.001,\ 0.005,\ 0.01$ and 0.1 respectively. For 1000 processors the maximum speed-up that can be achieved for a sequential portion of only 1% is less than 91. This rather depressing forecast put a serious damper on the possibilities of massive parallel implementations of algorithms and led Gustafson in 1988 to issue a counter claim [26]. Gustafson stated that a problem size is virtually never independent of the number of processors, as it appears in equation (2.2), but rather:

. . . in practice, the problem size scales with the number of processors.

Gustafson thus derives a maximum speed-up of:

$$\begin{aligned} \text{maximum speed-up} &= \frac{(s + (p \times n))}{s + p} \\ &= n + (1 - n) \times s \end{aligned} \tag{2.3}$$

This maximum speed-up according to Gustafson is also shown in figure 2.15. As the curve shows, the maximum achievable speed-up is nearly linear when the problem size is increased as more processing elements are added. Despite this optimistic forecast, Gustafson's premise is not applicable in a large number of cases. Most scientists and engineers have a particular problem they want to solve in as short a time as possible. Typically, the application already has a specified size for the problem domain. For example, in parallel radiosity we will be considering the diffuse lighting within a particular environment subdivided into a necessary number of patches. In this example it would be inappropriate for us to follow Gustafson's advice and increase the problem size as more processing elements were added to their parallel implementation, because to do so would mean either:

- the physical size of the three dimensional objects within the environmentwould have to be increased, which is of course not possible; or,
- the size of the patches used to approximate the surface would have to be reduced, thereby increasing the number of patches and thus the size of the problem domain.

This latter case is also not an option, because the computational method is sensitive to the size of the patches relative to their distances apart. Artificially significantly decreasing the size of the patches may introduce numerical instabilities into the method. Furthermore, artificially increasing the size of the problem domain may improve speed-up, but it *will not* improve the time taken to solve the problem.

For fixed sized problems it appears that we are left with Amdahl's gloomy prediction of the maximum speed-up that is possible for our parallel implementation. However, all is not lost, as Amdahl's assumption that an algorithm can be separated into a component which has to be executed sequentially and part which can be performed in parallel, may not be totally appropriate for the domain decomposition approach. Remember, in this model we are retaining the complete sequential algorithm and exploiting the parallelism that exists in the problem domain. So, in this case, an equivalent to Amdahl's law would imply that the data can be divided into two parts, that which must be dealt with in a strictly sequential manner and that which can executed in parallel. Any data dependencies will certainly imply some form of sequential ordering when dealing with the data, however, for a large number of problems such data dependencies may not exist. It may also be possible to reduce the effect of dependencies by clever scheduling.

The achievable speed-up for a problem using the domain decomposition approach is, however, bounded by the number of tasks that make up the problem. Solving a problem comprising a maximum of twenty tasks on more than twenty processors makes no sense. In practice, of course, any parallel implementation suffers from realisation penalties which increase as more processing elements are added. The actual speed-up obtained will thus be less than the maximum possible speed-up.

2.4.3 Efficiency

A relative efficiency based on the performance of the problem on one processor, can be a useful measure as to what percentage of a processor's time is being spent in useful computation. This, therefore, determines what the system overheads are. The relative efficiency we will measure as:

$$\text{Efficiency} = \frac{\text{speed-up} \times 100}{\text{number of processors}} \tag{2.4}$$

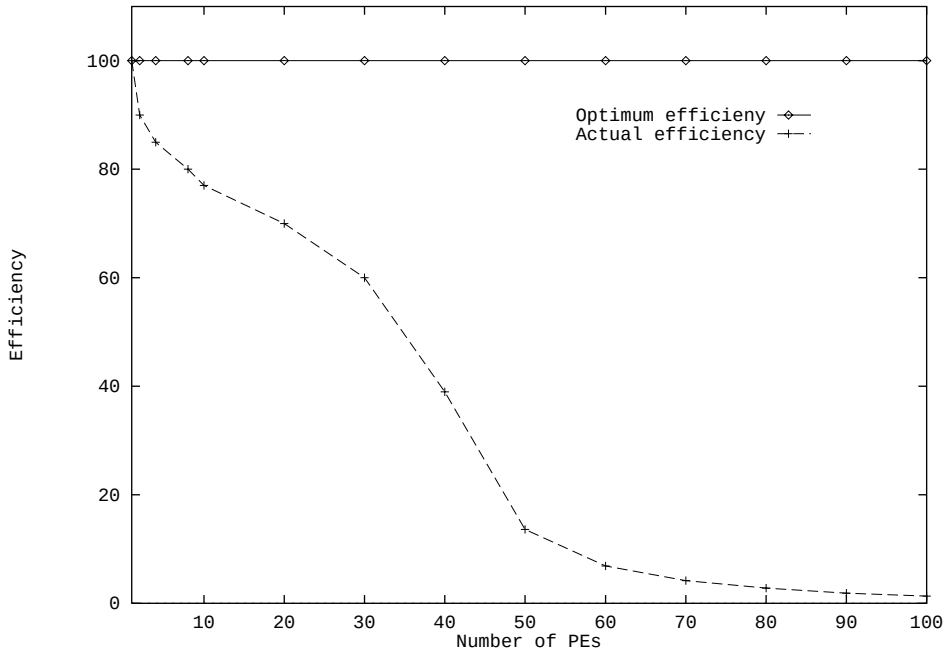


Figure 2.16: Optimum and actual processing element efficiency

Figure 2.16 shows the optimum and actual computation times given in figure 2.12 represented as processing element efficiency. The graph shows that optimum computation time, and therefore linear speed-up, equates to an efficiency of 100% for each processing element. This again shows that to achieve this level of efficiency every processing element must spend 100% of its time performing useful computation. Any implementation penalty would be immediately reflected by a decrease in efficiency. This is clearly shown in the curve for the actual computation times. Here the efficiency of each processing element decreases steadily as more are added until by the time 100 processing elements are incorporated, the realisation penalties are so high that each processing element is only able to devote just over 1% of its time to useful computation.

Optimum number of processing elements

Faced with implementing a fixed size problem on a parallel system, it may be useful to know the optimum number of processing elements on which this particular problem should be implemented in order to achieve the best possible performance. We term this optimum number n_{opt}. We shall judge the *maximum performance* for a particular problem

with a fixed problem domain size, as the shortest possible time required to produce the desired results for a certain parallel implementation. This optimum number of processing elements may be derived directly from the "computation time" graph. In figure 2.12 the minimum actual computation time occurred when the problem was implemented on 30 processing elements. As figure 2.17 shows, this optimum number of processing elements is also the point on the horizontal axis in figure 2.13 at which the maximum speed-up was obtained.

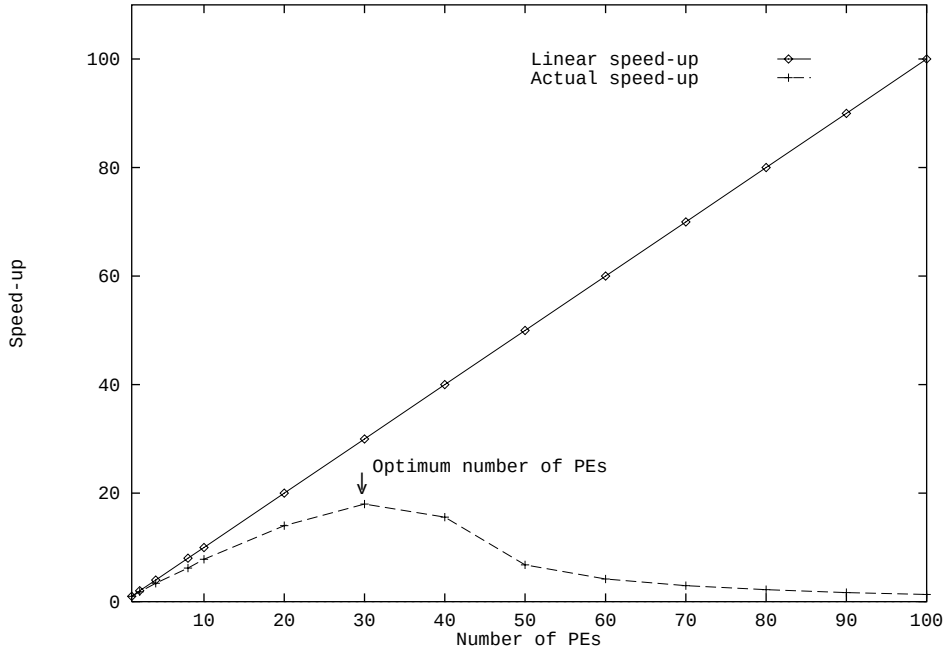


Figure 2.17: Optimum number of processing elements related to speed-up

The optimum number of processing elements is also the upper bound for the scalability of the problem for that parallel implementation. To improve the scalability of the problem it is necessary to re-examine the decisions concerning the algorithm chosen and the make-up of the system software that has been adopted for supporting the parallel implementation. As we will see in the subsequent chapters, the correct choice of system software can have a significant effect on the performance of a parallel implementation.

Figure 2.18 shows the speed-up graphs for different system software decisions for the *same* problem. The goal of a parallel implementation may be restated as:

> *"to ensure that the optimum number of processing elements for your problem is greater than the number of processing elements physically available to solve the problem!"*

Other metrics

Computation time, speed-up and efficiency provide insight into how successful a parallel implementation of a problem has been. As figure 2.18 shows, different implementations of the same algorithm on the same multiprocessor system may produce very different performances. A multitude of other metrics have been proposed over the years as a means of

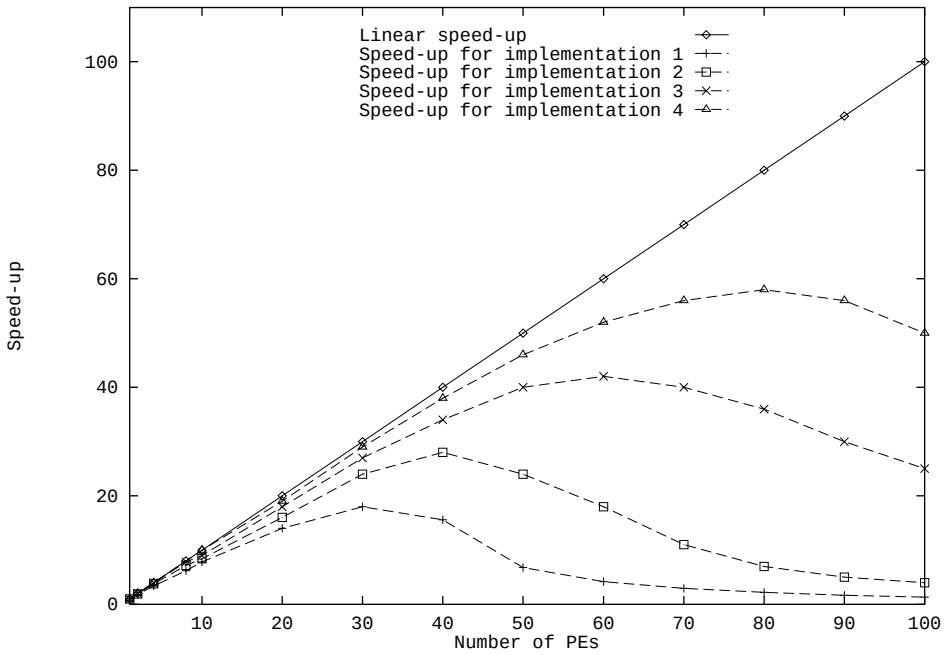


Figure 2.18: Speed-up graphs for different system software for the same problem

comparing the relative merits of different architectures and to provide a way of assessing their suitability as the chosen multiprocessor machine.

The performance of a computer is frequently measured as the rate of some number of events per second. Within a multi-user environment the elapsed time to solve a problem will comprise the user's CPU time plus the system's CPU time. Assuming that the computer's clock is running at a constant rate, the user's CPU performance may be measured as:

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{clock rate (eg. 100MHz)}}$$

The average clock cycles per instruction (CPI) may be calculated as:

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

We can also compute the CPU time from the time a program took to run:

$$\begin{aligned}\text{CPU time} &= \frac{\text{seconds}}{\text{program}} \\ &= \frac{\text{seconds}}{\text{clock cycle}} \times \frac{\text{clock cycles}}{\text{instructions}} \times \frac{\text{instructions}}{\text{program}}\end{aligned}$$

Such a performance metric is dependent on:

Clock rate: this is determined by the hardware technology and the organisation of the architecture;

CPI: a function of the system organisation and the instruction set architecture; and,

Instruction count: this is affected by the instruction set architecture and the compiler technology utilised.

One of the most frequently used performance metrics is the *MIPS* rating of a computer, that is how many Million Instructions Per Second the computer is capable of performing:

$$\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$$

However, the MIPS value is dependent on the instruction set used and thus any comparison between computers with different instruction sets is not valid. The MIPS value may even vary between programs running on the same computer. Furthermore, a program which makes use of hardware floating point routines may take *less time* to complete than a similar program which uses a software floating point implementation, but the first program will have a *lower* MIPS rating than the second [28]. These anomalies have led to MIPS sometimes being referred to as *"Meaningless Indication of Processor Speed"*.

Similar to MIPS is the "Mega-FLOPS" (MFLOPS) rating for computers, where MFLOPS represents Million FLoating point OPerations per Second:

$$\text{MFLOPS} = \frac{\text{no. of floating point operations in a program}}{\text{execution time} \times 10^6}$$

MFLOPS is not universally applicable, for example a word processor utilising no floating point operations would register no MFLOPS rating. However, the same program executing on different machines should be comparable, because, although the computers may execute a different number of instructions, they should perform the same number of operations, provided the set of floating point operations is consistent across both architectures. The MFLOPS value will vary for programs running on the same computer which have different mixtures of integer and floating point instructions as well as a different blend of "fast" and "slow" floating point instructions. For example, the *add* instruction often executes in less time than a *divide* instruction.

A MFLOPS rating for a single program can not, therefore, be generalised to provide a single performance metric for a computer. A suite of benchmark programs, such as the LINPACK or Livermore Loops routines, have been developed to allow a more meaningful method of comparison between machines. When examining the relative performance of computers using such benchmarks it is important to discover the *sustained* MFLOPS performance as a more accurate indication of the machines' potential rather than merely the *peak* MFLOPS rating, a figure that *"can be guaranteed never to be exceeded"*.

Other metrics for comparing computers include:

Dhrystone: A CPU intensive benchmark used to measure the integer performance especially as it pertains to system programming.

Whetstone: A synthetic benchmark without any vectorisable code for evaluating floating point performance.

TPS: Transactions Per Second measure for applications, such as airline reservation systems, which require on-line database transactions.

KLIPS: Kilo Logic Inferences Per Second is used to measure the relative inference performance of artificial intelligence machines

Tables showing the comparison of the results of these metrics for a number of architectures can be found in several books, for example [32, 37].

Cost is seldom an issue that can be ignored when purchasing a high performance computer. The desirability of a particular computer or even the number of processors within a system may be offset by the extraordinarily high costs associated with many high performance architectures. This prompted an early "law" by Grosch that the speed of a computer is proportional to its cost [25, 24]. Fortunately, although this is no longer completely true, multiprocessor machines are nevertheless typically more expensive than their general purpose counterparts. The parallel computer eventually purchased should provide acceptable computation times for an affordable price, that is maximise: "*the bangs per buck*" (performance per unit price).

Chapter 3

Task Scheduling

The efficient solution of a problem on a parallel system requires the computational performance of the processing elements to be fully utilised. Any processing element that is not busy performing useful computations is degrading overall system performance. Task scheduling strategies may be used to minimise these potential performance limitations.

3.1 Problem Decomposition

A problem may be solved on a parallel system by either exploiting the parallelism inherent in the algorithm, known as *algorithmic decomposition*, or by making use of the fact that the algorithm can be applied to different parts of the problem domain in parallel, which is termed *domain decomposition*. These two decomposition methods can be further categorised as shown in figure 3.1.

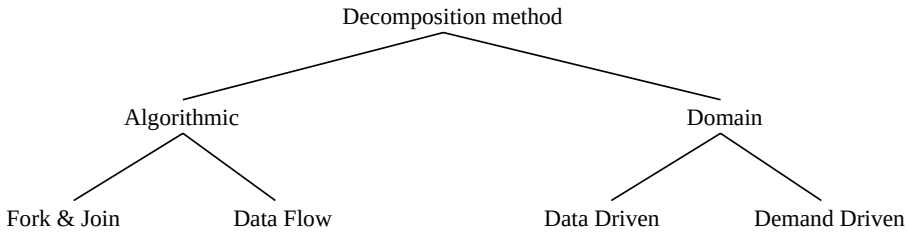


Figure 3.1: Methods of decomposing a problem to exploit parallelism

Over the years, an abundance of algorithms have been developed to solve a multitude of problems on sequential machines. A great deal of time and effort has been invested in the production of these sequential algorithms. Users are thus loathed to undertake the development of novel parallel algorithms, and yet still demand the performance that multiprocessor machines have to offer.

Algorithmic decomposition approaches to this dilemma have led to the development of compilers, such as those for High Performance Fortran, which attempt to parallelise automatically these existing algorithms. Not only do these compilers have to identify the parallelism hidden in the algorithm, but they also need to decide upon an effective strategy to place the identified segments of code within the multiprocessor system so that they can interact efficiently. This has proved to be an extremely hard goal to accomplish.

The domain decomposition approach, on the other hand, requires little or no modification to the existing sequential algorithm. There is thus no need for sophisticated compiler technology to analyse the algorithm. However, there will be a need for a parallel framework in the form of system software to support the division of the problem domain amongst the parallel processors.

3.1.1 Algorithmic decomposition

In algorithmic decomposition the algorithm itself is analysed to identify which of its features are capable of being executed in parallel. The finest granularity of parallelism is achievable at the operation level. Known as *dataflow*, at this level of parallelism the data "flows" between individual operands which are being executed in parallel [1]. An advantage of this type of decomposition is that little data space is required per processor [29], however, the communication overheads may be very large due to the very poor computation to communication ratio.

Fork & join parallelism, on the other hand, allocates portions of the algorithm to separate processors as the computation proceeds. These portions are typically several statements or complete procedures. The difference between the two algorithmic forms of decomposition is shown for a simple case in figure 3.2.

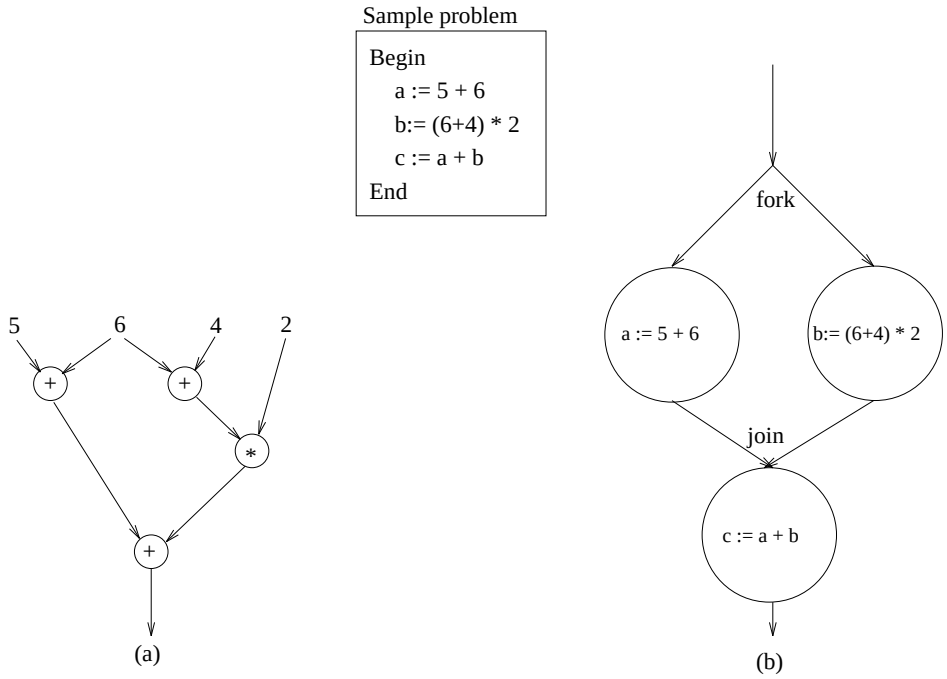


Figure 3.2: Algorithmic decomposition: (a) dataflow (b) fork & join

3.1.2 Domain decomposition

Instead of determining the parallelism inherent in the algorithm, domain decomposition examines the problem domain to ascertain the parallelism that may be exploited by solving the algorithm on distinct data items in parallel. Each parallel processor in this approach

will, therefore, have a complete copy of the algorithm and it is the problem domain that is divided amongst the processors. Domain decomposition can be accomplished using either a data driven or demand driven approach.

As we shall see, given this framework, the domain decomposition approach is applicable to a wide range of problems. Adoption of this approach to solve a particular problem in parallel, consists of two steps:

1. **Choosing the appropriate sequential algorithm.**

   Many algorithms have been honed over a number of years to a high level of perfection for implementation on sequential machines. The data dependencies that these highly sequential algorithms exhibit may substantially inhibit their use in a parallel system. In this case alternative sequential algorithms which are more suitable to the domain decomposition approach will need to be considered.

2. **Analysis of the problem in order to extract the criteria necessary to determine the optimum system software.**

   The system software provides the framework in which the sequential algorithm can execute. This system software takes care of ensuring each processor is kept busy, the data is correctly managed, and any communication within the parallel system is performed rapidly. To provide maximum efficiency, the system software needs to be tailored to the requirements of the problem. There is thus no *general purpose* parallel solution using the domain decomposition approach, but, as we shall see, a straightforward analysis of any problem's parallel requirements, will determine the correct construction of the system software and lead to an efficient parallel implementation.

Before commencing the detailed description of how we intend to tackle the solution of realistic rendering problems in parallel, it might be useful to clarify some of the terminology we shall be using.

3.1.3 Abstract definition of a task

The domain decomposition model solves a single problem in parallel by having multiple processors apply the same sequential algorithm to different data items from the problem domain in parallel. The lowest unit of computation within the parallel system is thus the application of the algorithm to one data item within the problem domain.

The data required to solve this unit of computation consists of two parts:

1. the *principal data items* (or PDIs) on which the algorithm is to be applied; and
2. *additional data items* (or ADIs) that may be needed to complete this computation on the PDIs.

For example, in ray tracing, we are computing the value at each pixel of our image plane. Thus these pixels would form our PDIs, while all the data describing the scene would constitute the ADIs. The problem domain is thus the pixels *plus* the scene description.

The application of the algorithm to a specified principal data item may be regarded as performing a single *task*. The task forms the elemental unit of computation within the parallel implementation. This is shown diagrammatically in figure 3.3.

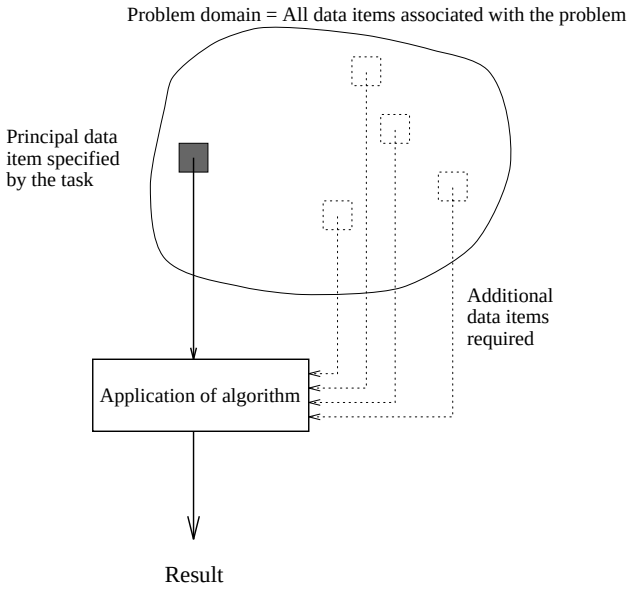


Figure 3.3: A task: the processing of a *principal* data item

3.1.4 System architecture

This tutorial is concentrating on implementing realistic rendering techniques on distributed memory systems (either a dedicated parallel machine or a distributed system of workstations). These processors may be connected together in some manner to form a *configuration*. A *process* is a segment of code that runs concurrently with other processes on a single processor. Several processes will be needed at each processor to implement the desired application and provide the necessary system software support. A *processing element* consists of a single processor together with these application and system processes and is thus the building block of the *multiprocessor system*. (We shall sometimes use the abbreviation *PE* for processing element in the figures and code segments.) When discussing configurations of processing elements, we shall use the term *links* to mean the communication paths between processes.

Structure of the system controller

To provide a useful parallel processing platform, a multiprocessor system must have access to input/output facilities. Most systems achieve this by designating at least one processing element as the *system controller* (SC) with the responsibilities of providing this input/output interface, as shown in figure 3.4. If the need for input/output facilities becomes a serious bottleneck then more than one system controller may be required. Other processing elements perform the actual computation associated with the problem.

In addition to providing the input/output facilities, the system controller may also be used to collect and collate results computed by the processing elements. In this case the system controller is in the useful position of being able to determine when the computation is complete and gracefully terminate the concurrent processes at every processing element.

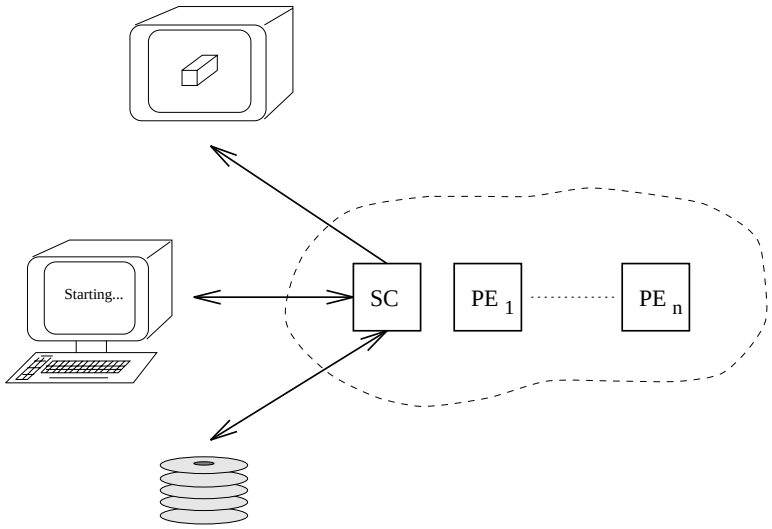


Figure 3.4: The system controller as part of a parallel system

3.2 Computational Models

The computational model chosen to solve a particular problem determines the manner in which work is distributed across the processors of the multiprocessor system. In our quest for an efficient parallel implementation we must maximise the proportion of time the processors spend performing necessary computation. Any imbalance may result in processors standing idle while others struggle to complete their allocated work, thus limiting potential performance. Load balancing techniques aim to provide an even division of computational effort to all processors.

The solution of a problem using the domain decomposition model involves each processing element applying the specified algorithm to a set of principal data items. The computational model ensures that every principal data item is acted upon and determines how the tasks are allocated amongst the processing elements. A choice of computation model exists for each problem. To achieve maximum system performance, the model chosen must see that the total work load is distributed evenly amongst the processing elements. This balances the overheads associated with communicating principal data items to processing elements with the need to avoid processing element idle time. A simplified ray tracing example illustrate the differences between the computational models.

A sequential solution to this problem may be achieved by dividing the image plane into twenty-four distinct regions, with each region constituting a single principal data item, as shown in figure 3.5, and then applying the ray tracing algorithm at each of these regions in turn. There are thus twenty-four tasks to be performed for this problem where each task is to compute the pixel value at one area of the image plane. To understand the computational models, it is not necessary to know the details of the algorithm suffice to say that each principal data item represents an area of the image plane on which the algorithm can be applied to determine the value forthat position. We will assume that no additional data items are required to complete any task.

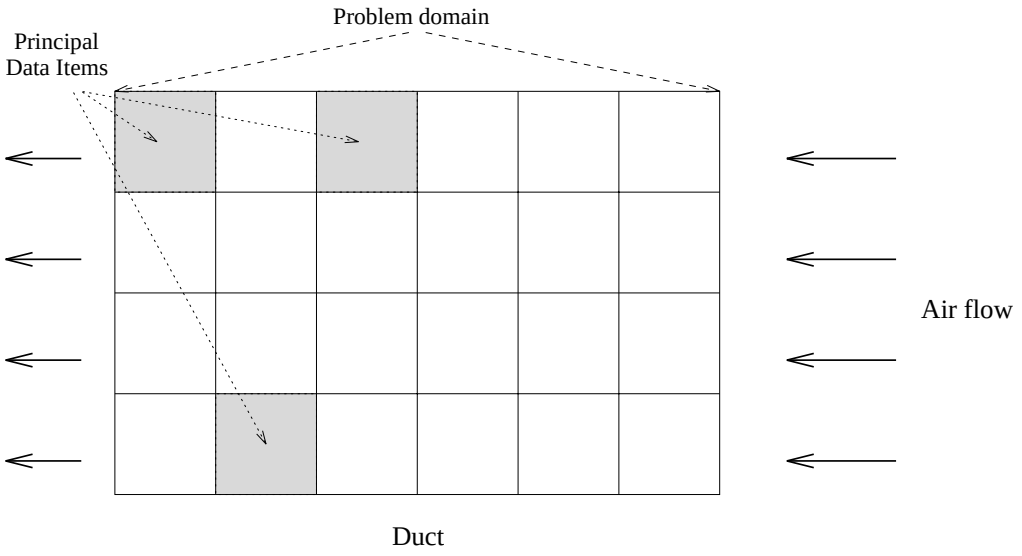


Figure 3.5: Principal data items for calculating the pixels in the image plane

3.2.1 Data driven model

The data driven model allocates all the principal data items to specific processing elements before computation commences. Each processing element thus knows *a priori* the principal data items to which they are required to apply the algorithm. Providing there is sufficient memory to hold the allocated set at each processing element, then, apart from the initial distribution, there is no further communication of principal data items. If there is insufficient local memory, then the extra items must be *fetched* as soon as memory space allows. This fetching of remote data items will be discussed further when data management is examined in Chapter 4.

Balanced data driven

In balanced data driven systems (also known as geometric decompositions), an equal number of principal data items is allocated to each processing element. This portion is determined simply by dividing the total number of principal data items by the number of processing elements:

$$\text{portion at each PE} = \frac{\text{number of principal data items}}{\text{number of PEs}}$$

If the number of principal data items is not an exact multiple of the number of processing elements, then

$$(number\ of\ principal\ data\ items)\ MOD\ (number\ of\ PEs)$$

will each have one extra principal data item, and thus perform one extra task. The required start task and the number of tasks is communicated by the system controller to each processing element and these can then apply the required algorithm to their allotted principal data items. This is similar to the way in which problems are solved on arrays of SIMD processors.

In this example, consider the simple ray tracing calculation for an empty scene. The principal data items (the pixels) may be allocated equally to three processing elements, labelled PE_1, PE_2 and PE_3, as shown in figure 3.6. In this case, each processing element is allotted eight principal data items.

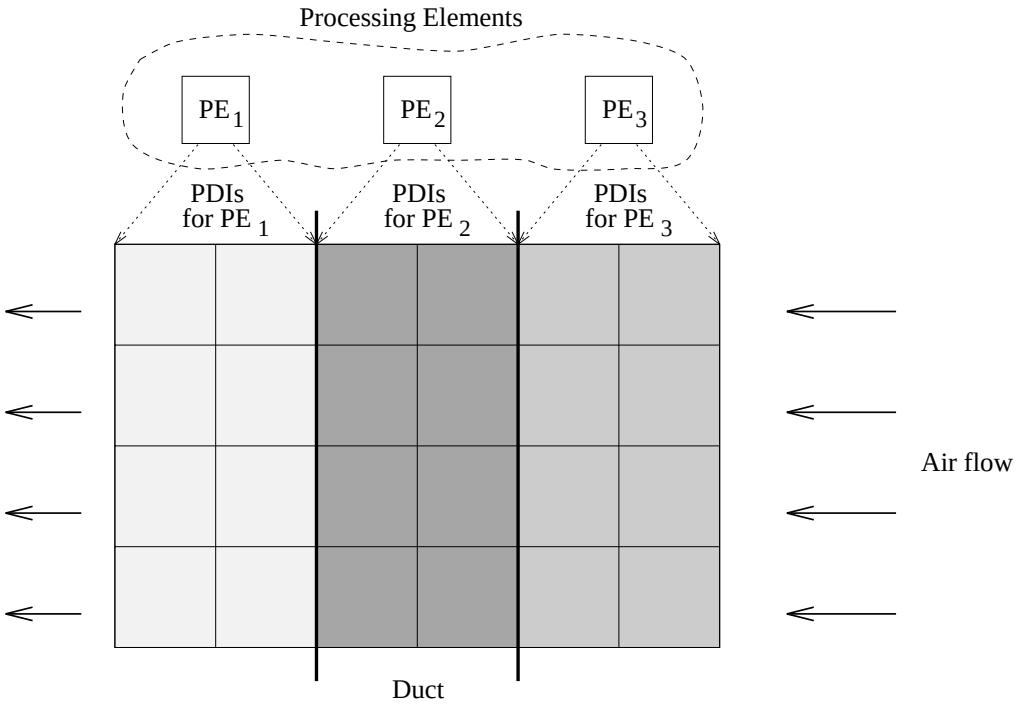


Figure 3.6: Equal allocation of data items to processing elements

As no further principal data item allocation takes place after the initial distribution, a balanced work load is only achieved for the balanced data driven computational model if the computational effort associated with each portion of principal data items is *identical*. If not, some processing elements will have finished their portions while others still have work to do. With the balanced data driven model the division of principal data items amongst processing elements is geometric in nature, that is each processing element simply may be allocated an equal number of principal data items irrespective of their position within the problem domain. Thus, to ensure a balanced work load, this model should only be used if the computational effort associated with each principal data item is the same, and preferably where the number of principal data items is an exact multiple of the number of processing elements. This implies *a priori* knowledge, but given this, the balanced data driven approach is the simplest of the computational models to implement.

Using figure 3.6, if the computation of each pixel 1 *time unit* to complete, then the sequential solution of this problem would take 24 *time units*. The parallel implementation of this problem using the three processing elements each allocated eight tasks should take approximately 8 *time units*, a third of the time required by the sequential implementation. Note, however, that the parallel solution will not be exactly one third of the sequential time as this would ignore the time required to communicate the portions from the system controller to the processing elements. This also ignores time required to receive the results back from the processing elements and for the system controller to collate the solution.

A balanced data driven version of this problem on the three processing elements would more accurately take:

$$\text{Solution time} = \text{initial distribution} + \lceil \frac{24}{3} \rceil + \text{result collation}$$

Assuming low communication times, this model gives the solution in approximately one third of the time of the sequential solution, close to the maximum possible linear speed-up. Solution of the same problem on five processing elements would give:

$$\text{Solution time} = \text{initial distribution} + \lceil \frac{24}{5} \rceil + \text{result collation}$$

This will be solved in even longer than the expected 4.8 *time units* as, in this case, one processing element is allocated 4 principal data items while the other four have to be apportioned 5. As computation draws to a close, one processing element will be idle while the four others complete their extra work. The solution time will thus be slightly more than 5 *time units*.

Unbalanced data driven

Differences in the computational effort associated with the principal data items will increase the probability of substantial processing element idle time if the simplistic balanced data driven approach is adopted. If the individual computation efforts differ, and are known *a priori*, then this can be exploited to achieve optimum load balancing.

The unbalanced data driven computational model allocates principal data items to processing elements based on their computational requirements. Rather than simply apportioning an equal number of tasks to each processing element, the principal data items are allocated to ensure that each processing element will complete its portion at approximately *the same time*.

For example, the complexity introduced into the ray tracing calculations by placing object into the scene, as shown in figure 3.7, will cause an increased computational effort required to solve the portions allocated to PE_1 and PE_2 in the balanced data driven model. This will result in these two processing elements still being busy with their computations long after the other processing element, PE_3, has completed its less computationally complex portion.

Should *a priori* knowledge be available regarding the computational effort associated with each principal data item then they may be allocated *unequally* amongst the processing elements, as shown in figure 3.8. The computational effort now required to process each of these unequal portions will be approximately the same, minimising any processing element idle time.

The sequential time required to solve the ray tracing with objects in the scene is now 42 *time units*. To balance the work load amongst the three processing elements, each processing element should compute for 14 *time units*. Allocation of the portions to each processing element in the unbalanced data driven model involves a preprocessing step to determine precisely the best way to subdivide the principal data items. The optimum compute time for each processing element can be obtained by simply dividing the total computation time by the number of processing elements. If possible, no processing element should be allocated principal data items whose combined computation time exceeds this optimum amount. Sorting the principal data items in descending computation times can facilitate the subdivision.

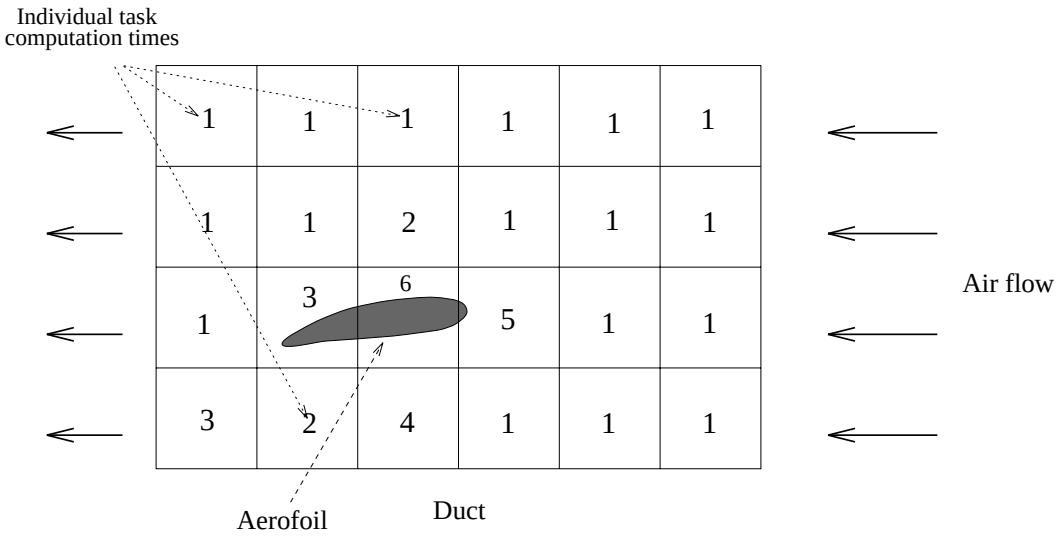


Figure 3.7: Unequal computational effort due to presence of objects in the scene

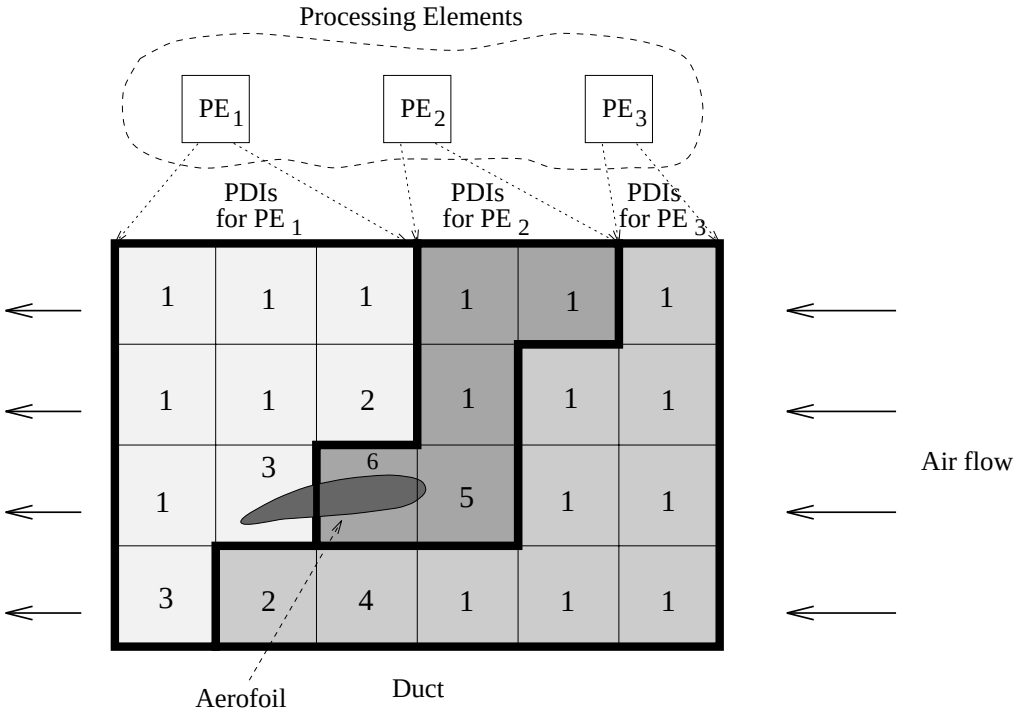


Figure 3.8: Unequal allocation of data items to processing elements to assist with load balancing

The total solution time for a problem using the unbalanced data driven model is thus:

$$\begin{aligned}\text{Solution time} &= \text{preprocessing} + \text{distribution} \\ &\quad + \text{longest portion time} + \text{result collation}\end{aligned}$$

So comparing the naive balanced distribution from section 3.2.1

$$\begin{aligned}\text{Balanced solution time} &= \text{distribution} + 21 + \\ &\quad \text{result collation}\end{aligned}$$

$$\begin{aligned}&\text{Unbalanced solution time} = \\ &\quad \text{preprocessing} + \text{distribution} + 14 + \text{result collation}\end{aligned}$$

The preprocessing stage is a simple sort requiring far less time than the ray tracing calculations. Thus, in this example, the unbalanced data driven model would be significantly faster than the balanced model due to the large variations in task computational complexity.

The necessity for the preprocessing stage means that this model will take more time to use than the balanced data driven approach should the tasks have the same computation requirement. However, if there are variations in computational complexity and they are known, then the unbalanced data driven model is the most efficient way of implementing the problem in parallel.

3.2.2 Demand driven model

The data driven computational models are dependent on the computational requirements of the principal data items being known, or at least being predictable, before actual computation starts. Only with this knowledge can these data items be allocated in the correct manner to ensure an even load balance. Should the computational effort of the principal data items be unknown or unpredictable, then serious load balancing problems can occur if the data driven models are used. In this situation the demand driven computational model should be adopted to allocate work to processing elements evenly and thus optimise system performance.

In the demand driven computational model, work is allocated to processing elements *dynamically* as they become idle, with processing elements no longer bound to any particular portion of the principal data items. Having produced the result from one principal data item, the processing elements demand the next principal data item from some work supplier process. This is shown diagrammatically in figure 3.9 for the simple ray tracing calculation.

Unlike the data driven models, there is no initial communication of work to the processing elements, however, there is now the need to send requests for individual principal data items to the supplier and for the supplier to communicate with the processing elements in order to satisfy these requests. To avoid unnecessary communication it may be possible to combine the return of the results from one computation with the request for the next principal data item.

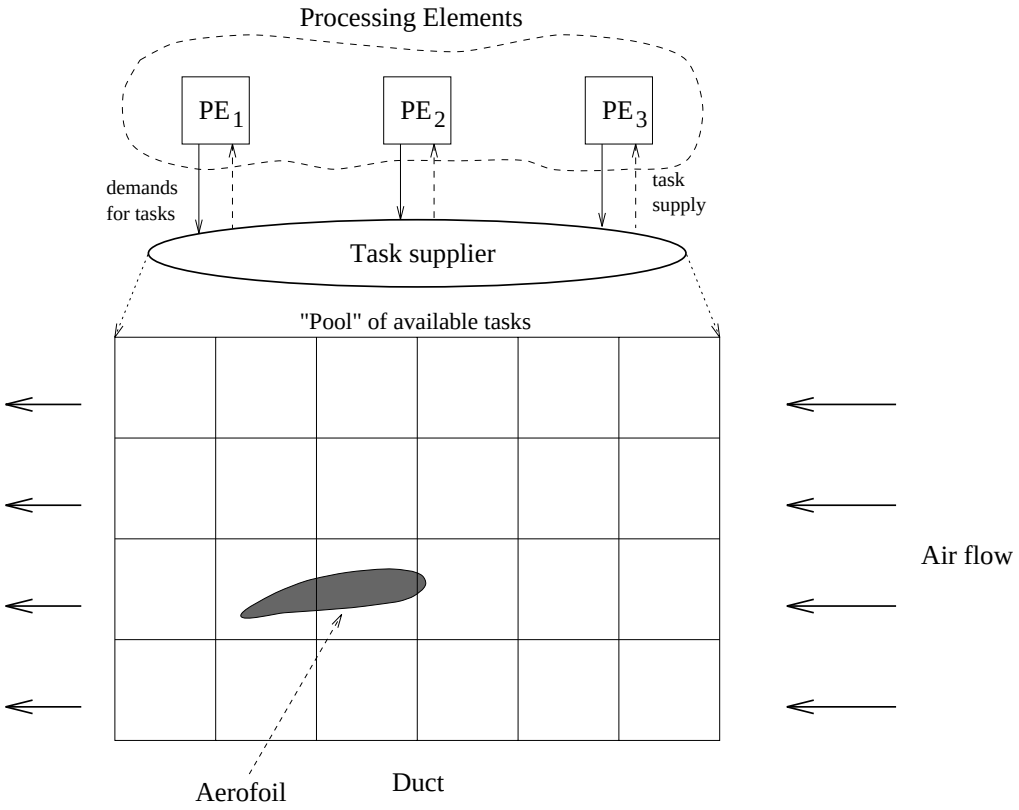


Figure 3.9: A demand driven model for a simple ray tracing calculation

The optimum time for solving a problem using this simple demand driven model is thus:

$$\text{Solution time} = 2 \times \text{total communication time} + \frac{\text{total computation time for all PDIs}}{\text{number of PEs}}$$

This optimum computation time, $\frac{\text{total computation time for all PDIs}}{\text{number of PEs}}$, will only be possible if the work can be allocated so that all processing elements complete the last of their tasks at exactly the same time. If this is not so then some processing elements will still be busy with their final task while the others have completed. It may also be possible to reduce the communication overheads of the demand driven model by overlapping the communication with the computation in some manner. This possibility will be discussed later in section 3.3.

On receipt of a request, if there is still work to be done, the work supplier responds with the next available task for processing. If there are no more tasks which need to be computed then the work supplier may safely ignore the request. The problem will be solved when all principal data items have been requested and all the results of the computations on these items have been returned and collated. The dynamic allocation of work by the demand driven model will ensure that while some processing elements are busy with more computationally demanding principal data items, other processing elements are available to compute the less complex parts of the problem.

Using the computational times for the presence of objects in the scene as shown in figure 3.8, figure 3.10 shows how the principal data items may be allocated by the task supplier to the processing elements using a simple serial allocation scheme. Note that the processing elements do not complete the same number of tasks. So, for example, while processing elements 2 and 3 are busy completing the computationally complex work associated with principal data items 15 and 16, processing elements 1 can compute the less computationally taxing tasks of principal data items 17 and 18.

The demand driven computational model facilitates dynamic load balancing when there is no prior knowledge as to the complexity of the different parts of the problem domain. Optimum load balancing is still dependent on all the processing elements completing the last of the work at the same time. An unbalanced solution may still result if a processing element is allocated a complex part of the domain towards the end of the solution. This processing element may then still be busy well after the other processing elements have completed computation on the remainder of the principal data items and are now idle as there is no further work to do. To reduce the likelihood of this situation it is important that the computationally complex portions of the domain, the so called *hot spots*, are allocated to processing elements early on in the solution process. Although there is no *a priori* knowledge as to the exact computational effort associated with any principal data item (if there were, an unbalanced data driven approach would have been adopted), nevertheless, any insight as to possible hot spot areas should be exploited. The task supplier would thus assign principal data items from these areas first.

In the ray tracing example , while the exact computational requirement associated with the principal data items in proximity of the objects in the scene may be unknown, it is highly likely that the solution of the principal items in that area will more complex than those elsewhere. In this problem, these principal data items should be allocated first.

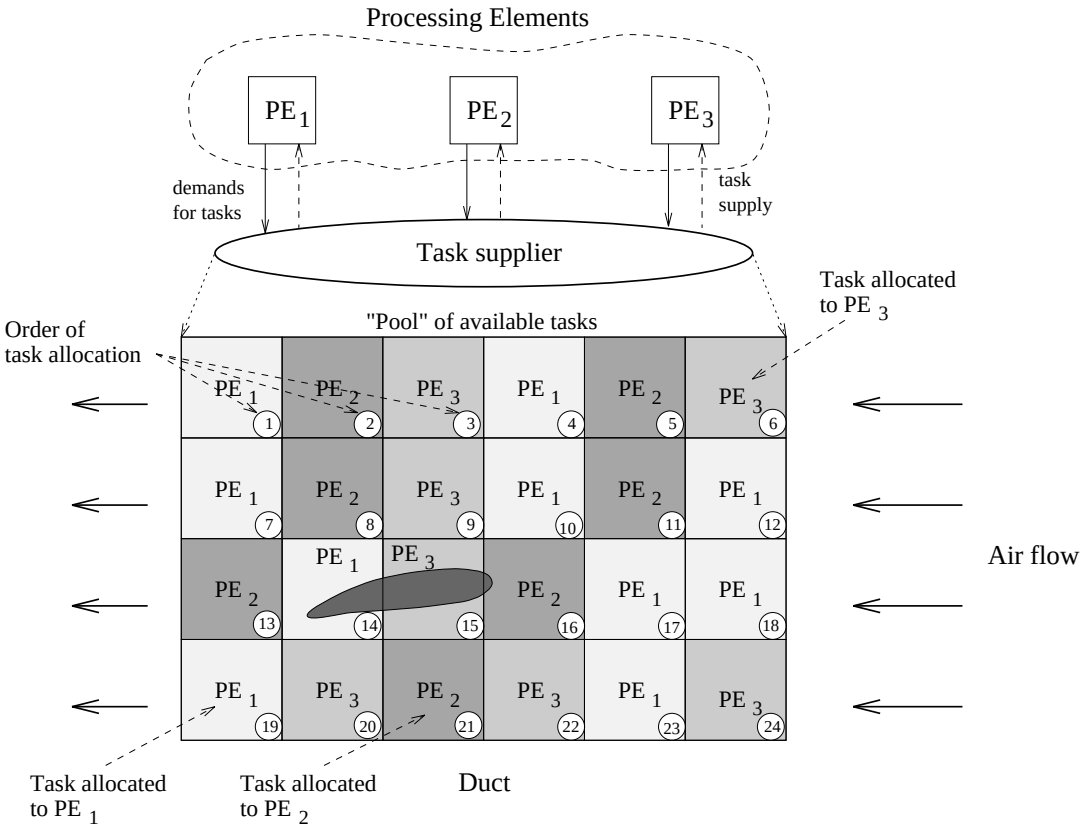


Figure 3.10: Allocation of principal data items using a demand driven model

If no insight is possible then a simple serial allocation, as shown in figure 3.10, or spiral allocation, as shown in figure 3.11 or even a random allocation of principal data items will have to suffice. While a random allocation offers perhaps a higher probability of avoiding late allocation of principal data items from hot spots, additional effort is required when choosing the next principal data item to allocate to ensure that no principal data item is allocated more than once.

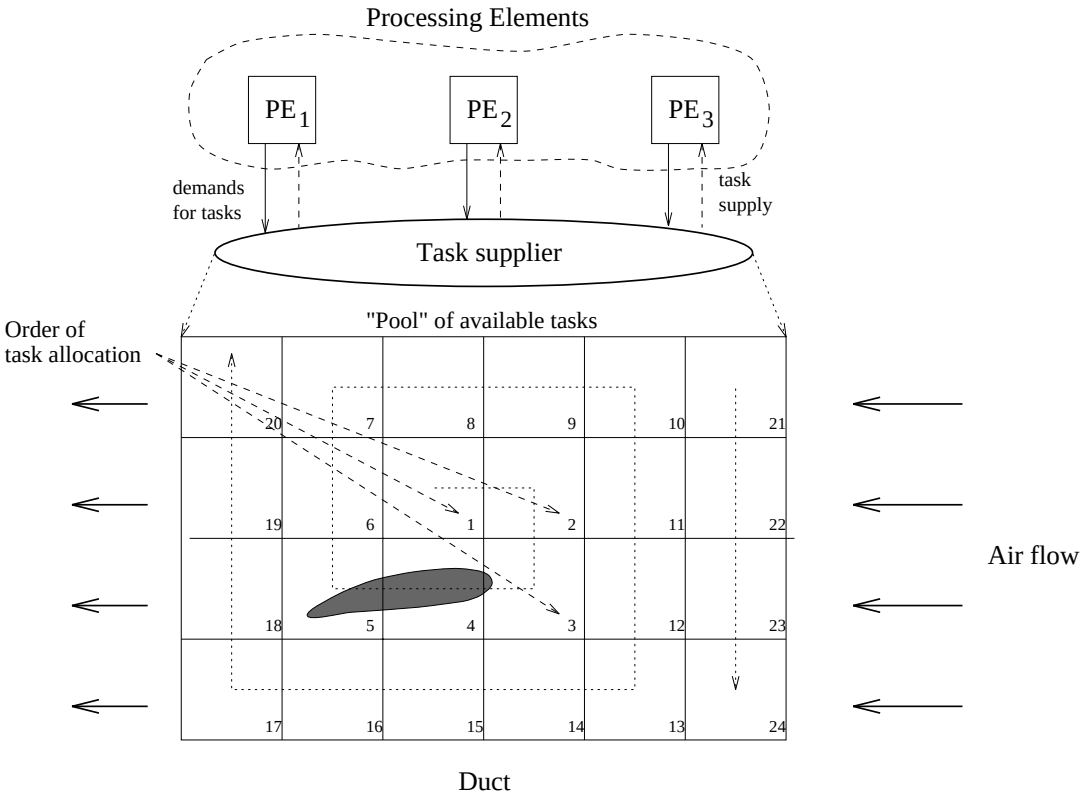


Figure 3.11: Allocation of principal data items in a spiral manner

As with all aspects of parallel processing, extra levels of sophistication can be added in order to exploit any information that becomes available as the parallel solution proceeds. Identifying possible hot spots in the problem domain may be possible from the computation time associated with each principal data item as these become known. If this time is returned along with the result for that principal data item, the work supplier can build a dynamic profile of the computational requirements associated with areas of the domain. This information can be used to adapt the allocation scheme to send principal data items from the possible hot spot regions. There is, of course, a trade off here between the possible benefits to load balancing in the early allocation of principal data items from hot spots, and the overhead that is introduced by the need to:

- time each computation at the processing elements;
- return this time to the work supplier;
- develop the time profile at the work supplier; and,
- adapt the allocation strategy to take into account this profile.

The benefits gained by such an adaptive scheme are difficult to predict as they are dependent on the problem being considered and the efficiency of the scheme implementation. The advice in these matters is always: "*implement a simple scheme initially and then add extra sophistication should resultant low system performance justify it.*"

3.2.3 Hybrid computational model

For most problems, the correct choice of computational model will either be one of the data driven strategies or the demand driven approach. However, for a number of problems, a hybrid computational model, exhibiting properties of both data and demand driven models, can be adopted to achieve improved efficiency. The class of problem that can benefit from the hybrid model is one in which an initial set of principal data items of known computational complexity may spawn an unknown quantity of further work.

In this case, the total number of principal data items required to solve the problem is *unknown* at the start of the computation, however, there are at least a known number of principal data items that must be processed first. If the computational complexity associated with these initial principal data items is unknown then a demand driven model will suffice for the whole problem, but if the computational complexity is known then one of the data driven models, with their lower communication overheads, should at least be used for these initial principal data items. Use of the hybrid model thus requires the computational model to be switched from data driven to demand driven mode as required.

3.3 Task Management

Task management encompasses the following functions:

- the definition of a task;
- controlling the allocation of tasks;
- distribution of the tasks to the processing elements; and,
- collation of the results, especially in the case of a problem with multiple stages.

3.3.1 Task definition and granularity

An *atomic element* may be thought of as a problem's lowest computational element within the sequential algorithm adopted to solve the problem. As introduced in section 3.1.2, in the domain decomposition model a single task is the application of this sequential algorithm to a principal data item to produce a result for the sub-parts of the problem domain. The task is thus the smallest element of computation for the problem within the parallel system. The *task granularity* (or *grain size*) of a problem is the number of atomic elements, which are included in one task. Generally, the task granularity remains constant for all tasks, but in some cases it may be desirable to alter dynamically this granularity as the computation proceeds. A task which includes only one atomic element is said to have the *finest granularity*, while a task which contains many is *coarser grained*, or has a *coarser granularity*. The actual definition of what constitutes a principal data item is determined by the granularity of the tasks.

A parallel system solves a problem by its constituent processing elements executing tasks in parallel. A *task packet* is used to inform a processing element which task, or

tasks, to perform. This task packet may simply indicate which tasks require processing by that processing element, thus forming the lowest level of distributed work. The packet may include additional information, such as additional data items, which the tasks require in order to be completed.

To illustrate the differences in this terminology, consider again the simple ray tracing problem. The atomic element of a sequential solution of this problem could be to perform a single ray-object intersection test. The principal data item is the pixel being computed and the additional data item required will be object being considered. A sequential solution of this problem would be for a single processing element to consider each ray-object intersection in turn. The help of several processing elements could substantially improve the time taken to perform the ray tracing.

The finest task granularity for the parallel implementation of this problem is for each task to complete one atomic element, that is perform one ray-object intersection. For practical considerations, it is perhaps more appropriate that each task should instead be to trace the complete path of a single ray. The granularity of each task is now the number of ray-object intersections required to trace this single ray and each pixel is a principal data item. A sensible task packet to distribute the work to the processing elements would include details about one or more pixels together with the necessary scene data (if possible, see Chapter 4).

To summarise our choices for this problem:

atomic element: to perform one ray-object intersection;

task: to trace the complete path of one ray (may consists of a number of atomic elements);

PDI: the pixel location for which we are computing the colour;

ADI: the scene data; and,

task packet: one or more rays to be computed.

Choosing the task granularity for the parallel implementation of a problem is not straightforward. Although it may be fairly easy to identify the atomic element for the sequential version of the problem, such a fine grain may not be appropriate when using many processing elements. Although the atomic element for ray tracing was specified as computing a single ray-object intersection in the above example, the task granularity for the parallel solution was chosen as computing the complete colour contribution at a particular pixel. If one atomic element had been used as the task granularity then additional problems would have introduced for the parallel solution, namely, the need for processors to to exchange partial results. This difficulty would have been exacerbated if, instead, the atomic element had been chosen as tracing a ray into a voxel and considering whether it does in fact intersect with an object there. Indeed, apart from the higher communication overhead this would have introduced, the issue of dependencies would also have to be checked to ensure, for example, that a ray was not checked against an object more than once.

As well as introducing additional communication and dependency overheads, the incorrect choice of granularity may also increase computational complexity variations and hinder efficient load balancing. The choice of granularity is seldom easy, however, a number of parameters of the parallel system can provide an indication as to the desirable granularity. The computation to communication ratio of the architecture will suggest whether additional communication is acceptable to avoid dependency or load balancing problems.

As a general rule, where possible, data dependencies should be avoided in the choice of granularity as these imply unnecessary synchronisation points within the parallel solution which can have a significant effect on overall system performance.

3.3.2 Task distribution and control

The task management strategy controls the distribution of packets throughout the system. Upon receipt, a processing element performs the tasks specified by a packet. The composition of the task packet is thus an important issue that must be decided before distribution of the tasks can begin. To complete a task a processing element needs a copy of the algorithm, the principal data item(s), and any additional data items that the algorithm may require for that principal data item. The domain decomposition paradigm provides each processing element with a copy of the algorithm, and so the responsibility of the task packet is to provide the other information.

The principal data items form part of the problem domain. If there is sufficient memory, it may be possible to store the entire problem domain as well as the algorithm at each processing element. In this case, the inclusion of the principal data item as part of the task packet is unnecessary. A better method would be simply to include the identification of the principal data item within the task packet. Typically, the identification of a principal data item is considerably smaller, in terms of actual storage capacity, than the item itself. The communication overheads associated with sending this smaller packet will be significantly less than sending the principal data item with the packet. On receipt of the packet the processing element could use the identification simply to fetch the principal data item from its local storage. The identification of the principal data item is, of course, also essential to enable the results of the entire parallel computation to be collated.

If the additional data items required by the task are known then they, or if possible, their identities, may also be included in the task packet. In this case the task packet would form an integral unit of computation which could be directly handled by a processing element. However, in reality, it may not be possible to store the whole problem domain at every processing element. Similarly, numerous additional data items may be required which would make their inclusion in the task packet impossible. Furthermore, for a large number of problems, the additional data items which are required for a particular principal data item may not be known in advance and will only become apparent as the computation proceeds.

A task packet should contain as a minimum either the identity, or the identity and actual principal data items of the task. The inability to include the other required information in the packet means that the parallel system will have to resort to some form of *data management*. This topic is described fully in Chapter 4.

3.3.3 Algorithmic dependencies

The algorithm of the problem may specify an order in which the work must be undertaken. This implies that certain tasks must be completed before others can commence. These dependencies must be preserved in the parallel implementation. In the worst case, algorithmic dependencies can prevent an efficient parallel implementation, as shown with the tower of toy blocks in figure 2.2. Amdahl's law, described in section 2.4, shows the implications to the algorithmic decomposition model of parallel processing of the presence of even a small percentage of purely sequential code. In the domain decomposition

approach, algorithmic dependencies may introduce two phenomena which will have to be tackled:

- *synchronisation points* which have the effect of dividing the parallel implementation into a number of distinct stages; and,
- *data dependencies* which will require careful data management to ensure a consistent view of the data to all processing elements.

Multi-stage algorithms

Many problems can be solved by a single stage of computation, utilising known principal data items to produce the desired results. However, the dependencies inherent in other algorithms may divide computation into a number of distinct stages. The *partial results* produced by one stage become the principal data items for the following stage of the algorithm, as shown in figure 3.12. For example, many scientific problems involve the construction of a set of simultaneous equations, a distinct stage, and the subsequent solution of these equations for the unknowns. The partial results, in this case elements of the simultaneous equations, become the principal data for the tasks of the next stage.

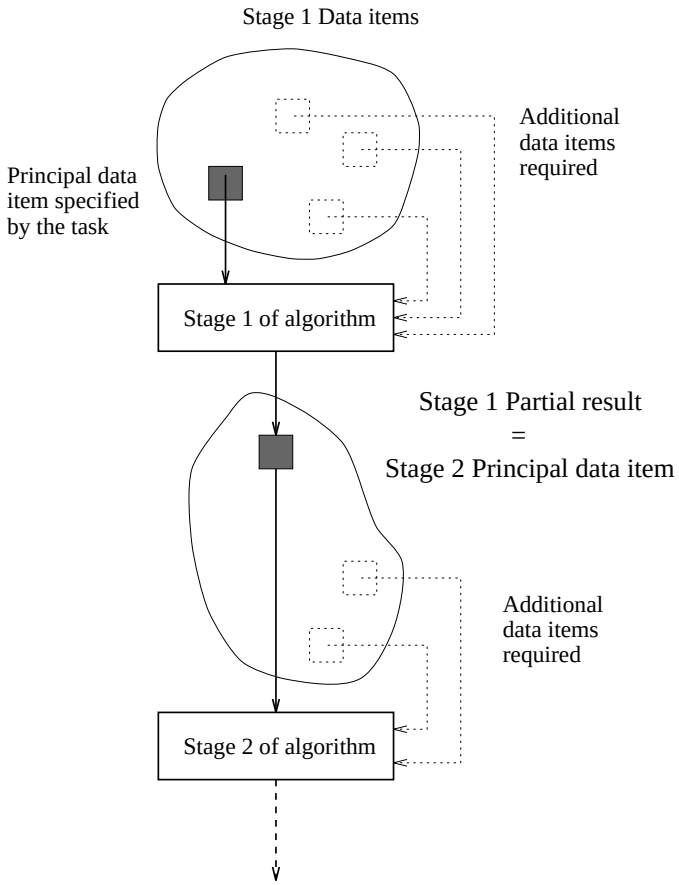


Figure 3.12: The introduction of partial results due to algorithmic dependencies

Even a single stage of a problem may contain a number of distinct substages which must first be completed before the next substage can proceed. An example of this is the

use of an iterative solver, such as the Jacobi method [22, 35], to solve a set of simultaneous equations. An iterative method starts with an approximate solution and uses it in a recurrence formula to provide another approximate solution. By repeatedly applying this process a sequence of solutions is obtained which, under suitable conditions, converges towards the exact solution.

Consider the problem of solving a set of six equations for six unknowns, $\mathbf{A}x = \mathbf{b}$. The Jacobi method will solve this set of equations by calculating, at each iteration, a new approximation from the values of the previous iteration. So the value for the x_i's at the n^{th} iteration are calculated as:

$$
\begin{aligned}
x_1^n &= \frac{b_i - a_{12}x_2^{n-1} - \ldots - a_{16}x_6^{n-1}}{a_{11}} \\
x_2^n &= \frac{b_i - a_{21}x_1^{n-1} - \ldots - a_{26}x_6^{n-1}}{a_{22}} \\
&\vdots \\
x_6^n &= \frac{b_i - a_{61}x_1^{n-1} - \ldots - a_{65}x_6^{n-1}}{a_{66}}
\end{aligned}
$$

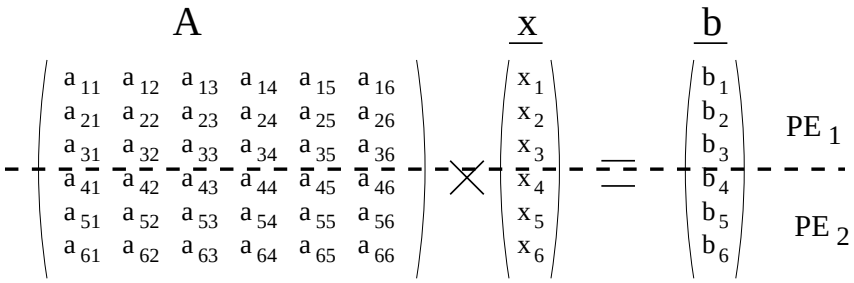


Figure 3.13: Solving an iterative matrix solution method on two processing elements

A parallel solution to this problem on two processing elements could allocate three rows to be solved to each processing element as shown in figure 3.13. Now PE_1 can solve the n^{th} iteration values $x_1^n,\ x_2^n$ and x_3^n in parallel with PE_2 computing the values of $x_4^n,\ x_5^n$ and x_6^n. However, neither processing element can proceed onto the $(n+1)^{st}$ iteration until both have finished the n^{th} iteration and exchanged their new approximations for the x_i^n's. Each iteration is, therefore, a substage which must be completed before the next substage can commence. This point is illustrated by the following code segment from PE_1:

```
PROCEDURE Jacobi() (* Executing on PE 1 *)
    Begin
        Estimate x[1] ... x[6]
        n := 0 (* Iteration number *)
        WHILE solution_not_converged DO
            Begin
                n := n + 1
                Calculate new x[1], x[2] & x[3] using old x[1] ... x[6]
                PARALLEL
                    SEND new x[1], x[2] & x[3] TO PE_2
                    RECEIVE new x[4], x[5] & x[6] FROM PE_2
            End
    End (* Jacobi *)
```

Data dependencies

The concept of dependencies was introduced in section 2.1.1 when we were unable to construct a tower of blocks in parallel as this required a strictly sequential order of task completion. In the domain decomposition model, data dependencies exist when a task may not be performed on some principal data item until another task has been completed. There is thus an implicit ordering on the way in which the task packets may be allocated to the processing elements. This ordering will prevent certain tasks being allocated, even if there are processing elements idle, until the tasks on which they are dependent have completed.

A linear dependency exists between each of the iterations of the Jacobi method discussed above. However, no dependency exists for the calculation of each x_i^n, for all i, as all the values they require, $x_j^{n-1},\ \forall j \neq i$, will already have been exchanged and thus be available at every processing element.

The Gauss-Seidel iterative method has long be preferred in the sequential computing community as an alternative to Jacobi. The Gauss-Seidel method makes use of new approximations for the x_i as soon as they are available rather than waiting for the next iteration. Provided the methods converge, Gauss-Seidel will converge more rapidly than the Jacobi method. So, in the example of six unknowns above, in the n^{th} the value of x_1^n would still be calculated as:

$$x_1^n = \frac{b_i - a_{12}x_2^{n-1} - \ldots - a_{16}x_6^{n-1}}{a_{11}},$$

but the x_2^n value would now be calculated by:

$$x_2^n = \frac{b_i - a_{21}\mathbf{x_1^n} - a_{23}x_3^{n-1} - \ldots - a_{26}x_6^{n-1}}{a_{22}}$$

Although well suited to sequential programming, the strong linear dependency that has been introduced, makes the Gauss-Seidel method poorly suited for parallel implementation. Now within each iteration no value of x_i^n can be calculated until all the values for $x_j^n,\ j < i$ are available; a strict sequential ordering of the tasks. The less severe data dependencies within the Jacobi method thus make it a more suitable candidate for parallel processing than the Gauss-Seidel method which is more efficient on a sequential machine.

It is possible to implement a hybrid of these two methods in parallel, the so-called "Block Gauss-Seidel - Global Jacobi" method. A processing element which is computing several rows of the equations, may use the Gauss-Seidel method for these rows as they will

be computed sequentially within the processing element. Any values for x_i^n not computed locally will assume the values of the previous iteration, as in the Jacobi method. All new approximations will be exchanged at each iteration. So, in the example, PE_2 would calculate the values of x_4^n, x_5^n and x_6^n as follows:

$$x_4^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{15}x_5^{n-1} - a_{16}x_6^{n-1}}{a_{44}}$$

$$x_5^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{14}\mathbf{x_4^n} - a_{16}x_6^{n-1}}{a_{55}}$$

$$x_6^n = \frac{b_i - a_{11}x_1^{n-1} - a_{12}x_2^{n-1} - a_{13}x_3^{n-1} - a_{14}\mathbf{x_4^n} - a_{15}\mathbf{x_5^n}}{a_{66}}$$

3.4 Task Scheduling Strategies

3.4.1 Data driven task management strategies

In a data driven approach, the system controller determines the allocation of tasks prior to computation proceeding. With the unbalanced strategy, this may entail an initial sorting stage based on the known computational complexity, as described in section 3.2.1. A single task-packet detailing the tasks to be performed is sent to each processing element. The application processes may return the results upon completion of their allocated portion, or return individual results as each task is performed, as shown in this code segment:

```
PROCESS Application_Process()
    Begin
        RECEIVE task_packet FROM SC via R
        FOR i = start_task_id TO finish_task_id DO
            Begin
                result[i] := Perform_Algorithm(task[i])
                SEND result[i] TO SC via R
            End
    End (* Application_Process *)
```

In a data driven model of computation a processing element may initially be supplied with as many of its allocated principal data items as its local memory will allow. Should there be insufficient storage capacity a simple data management strategy may be necessary to prefetch the missing principal data items as computation proceeds and local storage allows. This is discussed further when considering the management of data in Chapter 4.

3.4.2 Demand driven task management strategies

Task management within the demand driven computational model is explicit. The work supplier process, which forms part of the system controller, is responsible for placing the tasks into packets and sending these packets to requesting processing elements. To facilitate this process, the system controller maintains a *pool* of already constituted task packets. On receipt of a request, the work supplier simply dispatches the next available task packet from this task pool, as can be seen in figure 3.14.

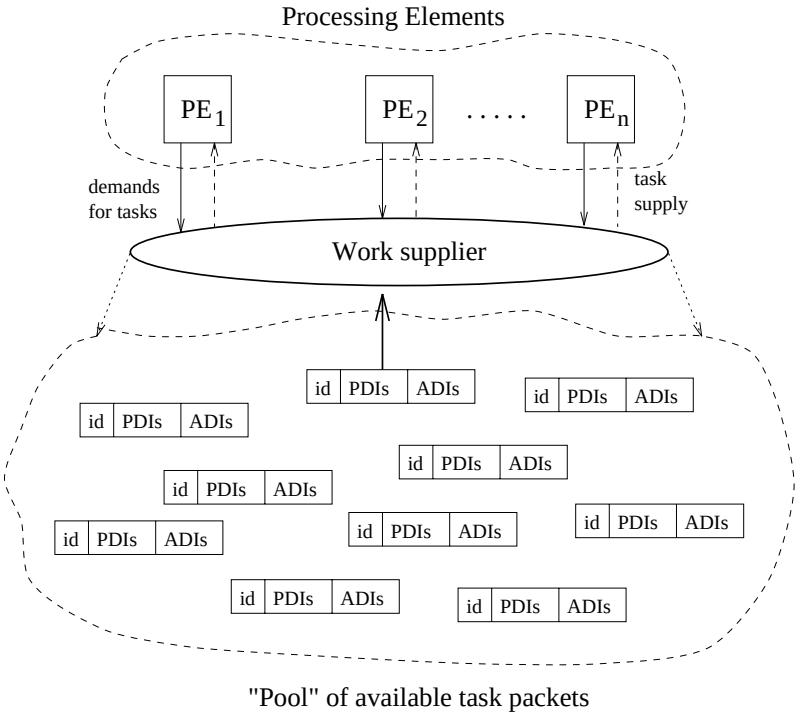


Figure 3.14: Supplying task packets from a task pool at the system controller

The advantage of a task pool is that the packets can be inserted into it in advance, or concurrently as the solution proceeds, according to the allocation strategy adopted. This is especially useful for problems that create work dynamically, such as those using the hybrid approach as described in section 3.2.3. Another advantage of the task pool is that if a hot spot in the problem domain is identified, then the ordering within the task pool can be changed dynamically to reflect this and thus ensure that potentially computationally complex tasks are allocated first.

More than one task pool may be used to reflect different levels of task priority. High priority tasks contained in the appropriate task pool will always be sent to a requesting processing element first. Only once this high priority task pool is (temporarily) empty will tasks from lower priority pools be sent. The multiple pool approach ensures that high priority tasks are not ignored as other tasks are allocated.

In the demand driven computational model, the processing elements demand the next task as soon as they have completed their current task. This demand is translated into sending a request to the work supplier, and the demand is only satisfied when the work supplier has delivered the next task. There is thus a definite delay period from the time the request is issued until the next task is received. During this period the processing element will be computationally idle. To avoid this idle time, it may be useful to include a buffer at each processing element capable of holding at least one task packet. This buffer may be considered as the processing element's own private task pool. Now, rather than waiting for a request to be satisfied from the remote system controller, the processing element may proceed with the computation on the task packet already present locally. When the remote request has been satisfied and a new task packet delivered, this can be stored in

the buffer waiting for the processing element to complete the current task.

Whilst avoiding delays in fetching tasks from a remote task pool, the use of a buffer at each processing element may have serious implications for load balancing, especially towards the end of the problem solution. We will examine these issues in more detail after we have considered the realisation of task management for a simple demand driven system - the processor farm.

A first approach: The processor farm

Simple demand driven models of computation have been implemented and used for a wide range of applications. One realisation of such a model, often referred to in the literature, is that implemented by May and Shepherd [47]. This simple demand driven model, which they term a *processor farm*, has been used for solving problems with high computation to communication ratios. The model proposes a single system controller and one or more processing elements connected in a linear configuration, or chain. The structure of a processing element in this model is shown in figure 3.15.

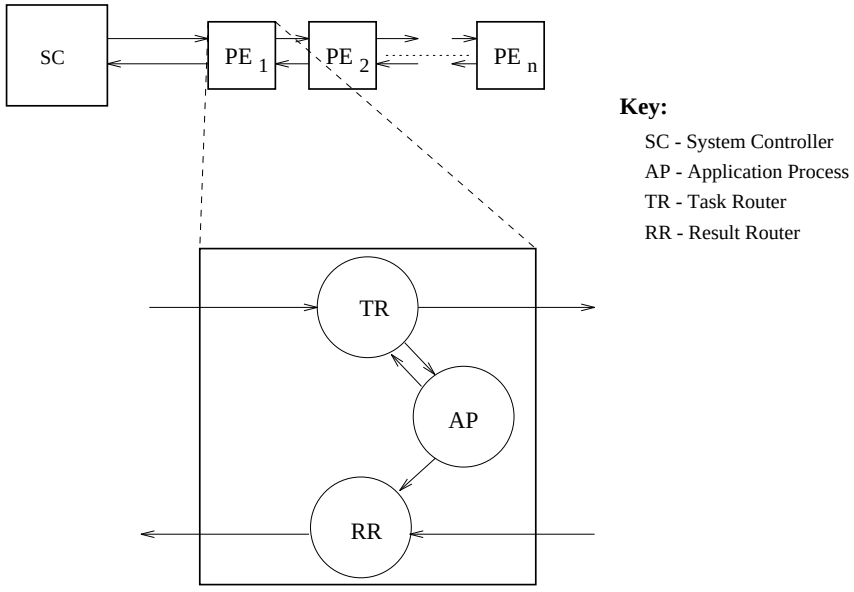


Figure 3.15: A processing element for the processor farm model

The application process performs the desired computation, while the communication within the system is dealt with by two router processes, the Task Router (TR) and the Result Router (RR). As their names suggest, the task router is responsible for distributing the tasks to the application process, while the result router returns the results from the completed tasks back to the system controller. The system controller contains the initial pool of tasks to be performed and collates the results. Such a communication strategy is simple to implement and largely problem independent.

To reduce possible processing element idle time, each task router process contains a single buffer in which to store a task so that a new task can be passed to the application process as soon as it becomes idle. When a task has been completed the results are sent to the system controller. On receipt of a result, the system controller releases a new task into

the system. This synchronised releasing of tasks ensures that there are never more tasks in the system than there is space available.

On receipt of a new task, the task router process either:

1. passes the task directly to the application process if it is waiting for a task; or
2. places the task into its buffer if the buffer is empty; or, otherwise
3. passes the task onto the next processing element in the chain.

The processor farm is initialised by loading sufficient tasks into the system so that the buffer at each task router is full and each application process has a task with which to commence processing. Figure 3.16 shows the manner in which task requests are satisfied within a simple two processing element configured in a chain.

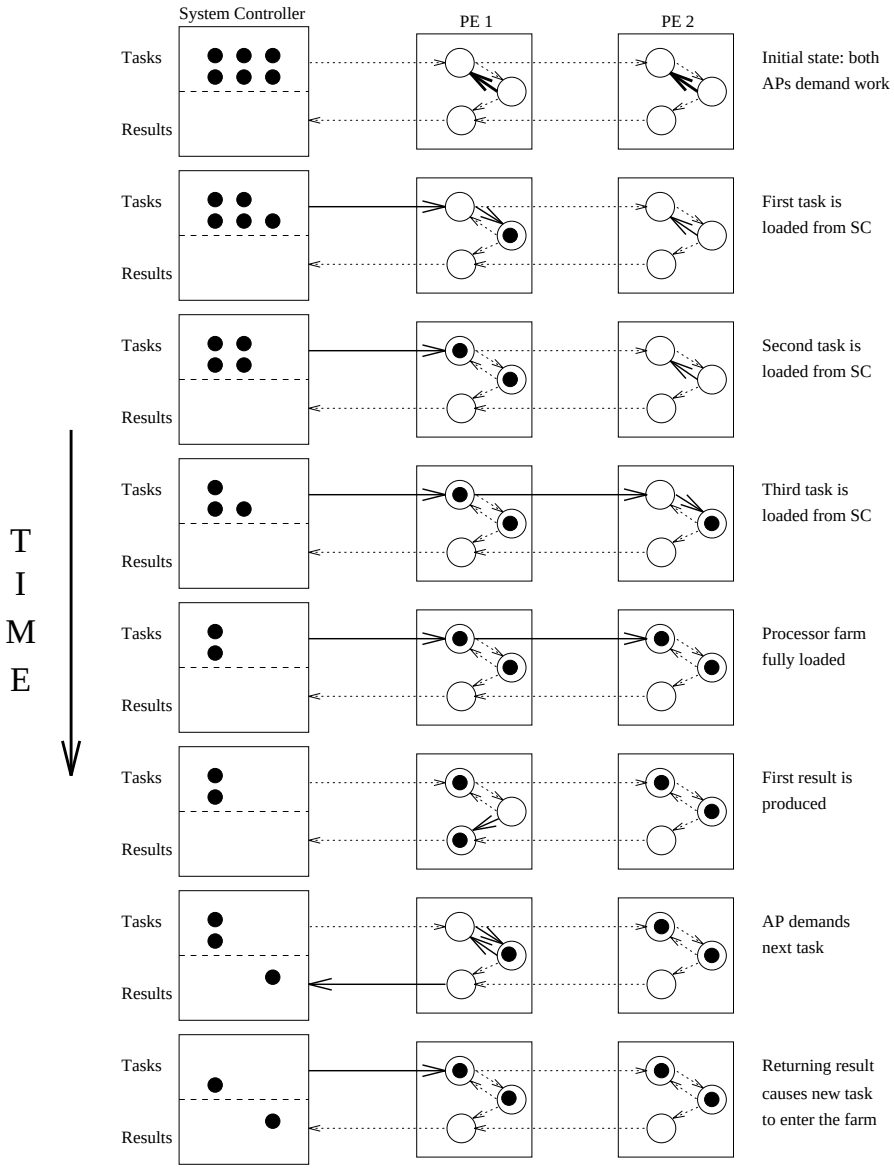


Figure 3.16: Task movement within a two PE processor farm

The simplicity of this realisation of a demand driven model has contributed largely to its popularity. Note that because of the balance maintained within the system, the only instance at which the last processing element is different from any other processing element in the chain is to ensure the `closedown_command` does not get passed any further. However, such a model does have disadvantages which may limit its use for more complex problems.

The computation to communication ratio of the desired application is critical in order to ensure an adequate performance of a processor farm. If this ratio is too low then significant processing element idle time will occur. This idle time occurs because the computation time for the application process to complete its current task and the task buffered at the task router may be lower than the combined communication time required for the results to reach the system controller plus the time for the new tasks released into the system to reach the processing element. This problem may be partially alleviated by the inclusion of several buffers at each task router instead of just one. However, without *a priori* knowledge as to the computation to communication ratio of the application, it may be impossible to determine precisely what the optimum number of buffers should be. This analysis is particularly difficult if the computational complexity of the tasks vary; precisely the type of problem demand driven models are more apt at solving. The problem independence of the system will also be compromised by the use of any *a priori* knowledge.

If the number of buffers chosen is too small, then the possibility of application process idle time will not be avoided. Provision of too many buffers will certainly remove any immediate application process idle time, but will re-introduce the predicament as the processing draws to a close. This occurs once the system controller has no further tasks to introduce into the system and now processing must only continue until all tasks still buffered at the processing elements have been completed. Obviously, significant idle time may occur as some processing elements struggle to complete their large number of buffered tasks.

The computation to communication ratio of the processor farm is severely exacerbated by the choice of the chain topology. The distance between the furthest processing element in the chain and the system controller grows linearly as more processing elements are added. This means that the combined communication time to return a result and receive a new task also increases. Furthermore, this communication time will also be adversely affect by the message traffic of all the intermediate processing elements which are closer to the system controller.

3.4.3 Task manager process

The aim of task management within a parallel system is to ensure the efficient supply of tasks to the processing elements. A Task Manager process (TM) is introduced at each processing element to assist in maintaining a continuous supply of tasks to the application process. The application process no longer deals with task requests directly, but rather indirectly using the facilities of the task manager. The task manager process assumes the responsibility for ensuring that every request for additional tasks from the application process will be satisfied immediately. The task manager attempts to achieve this by maintaining a local task pool.

In the processor farm, the task router process contains a single buffered task in order to satisfy the next local task request. As long as this buffer is full, task supply is immediate

as far as the application process is concerned. The buffer is refilled by a new task from the system controller triggered on receipt of a result. The task router acts in a *passive* manner, awaiting replenishment by a new task within the farm. However, if the buffer is empty when the application process requests a task then this process must remain idle until a new task arrives. This idle time is wasted computation time and so to improve system performance the passive task router should be replaced by a "intelligent" task manager process more capable of ensuring new tasks are always available locally.

The task management strategies implemented by the task manager and outlined in the following sections are *active*, dynamically requesting and acquiring tasks during computation. The task manager thus assumes the responsibility of ensuring local availability of tasks. This means that an application process should *always* have its request for a task satisfied immediately by the task manager unless:

- at the start of the problem the application processes make a request before the initial tasks have been provided by the system controller;
- there are no more tasks which need to be solved for a particular stage of the parallel implementation; or,
- the task manager's replenishment strategy has failed in some way.

A local task pool

To avoid any processing element idle time, it is essential that the task manager has at least one task available locally at the moment the application process issues a task request. This desirable situation was achieved in the processor farm by the provision of a single buffer at each task router. As we saw, the single buffer approach is vulnerable to the computation to communication ratio within the system. Adding more buffers to the task router led to the possibility of serious load imbalances towards the end of the computation.

The task manager process maintains a local task pool of tasks awaiting computation by the application process. This pool is similar to the task pool at the system controller, as shown in figure 3.14. However, not only will this local pool be much smaller than the system controller's task pool, but also it may be desirable to introduce some form of "status" to the number of available tasks at any point in time.

Satisfying a task request will free some space in the local task pool. A simple replenishment strategy would be for the task manager immediately to request a new task packet from the system controller. This request has obvious communication implications for the system. If the current message densities within the system are high and as long as there are still tasks available in the local task pool, this request will place and unnecessary additional burden on the already overloaded communication network.

As an active process, it is quite possible for the task manager to delay its replenishment request until message densities have diminished. However, this delay must not be so large that subsequent application process demands will deplete the local task pool before any new tasks can be fetched causing processor idle time to occur. There are a number of indicators which the task manager can use to determine a suitable delay. Firstly, this delay is only necessary if current message densities are high. Such information should be available for the router. Given a need for delay, the number of tasks in the task pool, the approximate computation time each of these tasks requires, and the probable communication latency in replenishing the tasks should all contribute to determining the request delay.

In a demand driven system, the computational complexity variations of the tasks are not known. However, the task manager will be aware of how long previous tasks have taken to compute (the time between application process requests). Assuming some form of preferred biased allocation of tasks in which tasks from similar regions of the problem domain are allocated to the same processing element, as discussed in section 3.4.5, the task manager will be able to build up a profile of task completion time which can be used to predict approximate completion times for tasks in the task pool. The times required to satisfy previous replenishment requests will provide the task manager with an idea of likely future communication responses. These values are, of course, mere approximations, but they can be used to assist in determining reasonable tolerance levels for the issuing of replenishment requests.

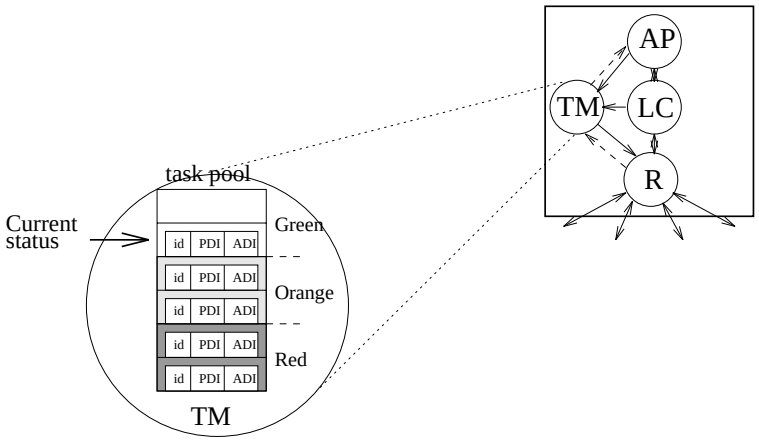


Figure 3.17: Status of task manager's task pool

The task manager's task pool is divided into three regions: *green*, *orange* and *red*. The number of tasks available in the pool will determine the current status level, as shown in figure 3.17. When faced with the need to replenish the task pool the decision can be taken based on the current status of the pool:

green: Only issue the replenishment request if current message traffic density is low;

orange: Issue the replenishment request unless the message density is very high; and,

red: Always issue the replenishment request.

The boundaries of these regions may be altered dynamically as the task manager acquires more information. At the start of the computation the task pool will be all red. The computation to communication ratio is critical in determining the boundaries of the regions of the task pool. The better this ratio, that is when computation times are high relative to the time taken to replenish a task packet, the smaller the red region of the task pool need be. This will provide the task manager with greater flexibility and the opportunity to contribute to minimising communication densities.

3.4.4 Distributed task management

One handicap of the centralised task pool system is that all replenishment task requests from the task managers must reach the system controller before the new tasks can be

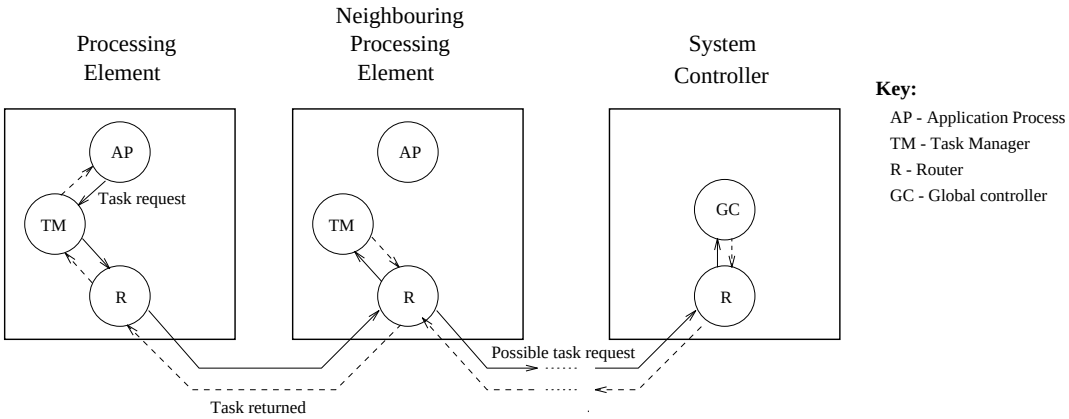


Figure 3.18: Task request propagating towards the system controller

allocated. The associated communication delay in satisfying these requests can be significant. The communication problems can be exacerbated by the bottleneck arising near the system controller. Distributed task management allows task requests to be handled at a number of locations remote from the system controller. Although all the tasks originate from the system controller, requests from processing elements no longer have to reach there in order to be satisfied.

The closest location for a task manager to replenish a task packet is from the task pool located at the task manager of one of its nearest neighbours. In this case, a replenishment request no longer proceeds directly to the system controller, but simply via the appropriate routers to the neighbouring task manager. If this neighbouring task manager is able to satisfy the replenishment request then it does so from its task pool. This task manager may now decide to in turn replenish its task pool, depending on its current status and so it will also request another task from one of its neighbouring task managers, but obviously not the same neighbour to which it has just supplied the task. One sensible strategy is to propagate these requests in a "chain like" fashion in the direction towards the main task supplier at the system controller, as shown in figure 3.18.

This distributed task management strategy is referred to as a *producer-consumer* model. The application process is the initial consumer and its local task manager the producer. If a replenishment request is issued then this task manager becomes the consumer and the neighbouring task manager the producer, and so on. The task supplier process of the system controller is the overall producer for the system. If no further tasks exist at the system controller then the last requesting task manager may change the direction of the search. This situation may occur towards the end of a stage of processing and facilitates load balancing of any tasks remaining in task manager buffers. As well as reducing the communication distances for task replenishment, an additional advantage of this "chain reaction" strategy is that the number of request messages in the system is reduced. This will play a major rôle helping maintain a lower overall message density within the system.

If a task manager is unable to satisfy a replenishment request as its task pool is empty, then to avoid "starvation" at the requesting processing element, this task manager must ensure that the request is passed on to another processing element.

A number of variants of the producer-consumer model are also possible:

- Instead of following a path towards the system controller, the "chain reaction" could follow a predetermined Hamiltonian path (the system controller could be one of the processors on this path).

  Aside: A Hamiltonian path is a circuit starting and finishing at one processing element. This circuit passes through each processor in the network once only.

  Such a path would ensure that a processing element would be assured of replenishing a task if there was one available and there would be no need to keep track of the progress of the "chain reaction" to ensure no task manager was queried more than once per chain.

- In the course of its through-routing activities a router may handle a task packet destined for a distant task manager. If that router's local task manager has an outstanding "red request" for a task then it is possible for the router to *poach* the "en route task" by diverting it, so satisfying its local task manager immediately. Care must be taken to ensure that the task manager for whom the task was intended is informed that the task has been poached, so it may issue another request. In general, tasks should only be poached from "red replenishment" if to do so would avoid local application process idle time.

3.4.5 Preferred bias task allocation

The preferred bias method of task management is a way of allocating tasks to processing elements which combines the simplicity of the balanced data driven model with the flexibility of the demand driven approach. To reiterate the difference in these two computational models as they pertain to task management:

- Tasks are allocated to processing elements in a predetermined manner in the balanced data driven approach.

- In the demand driven model, tasks are allocated to processing elements on demand. The requesting processing element will be assigned the next available task packet from the task pool, and thus no processing element is bound to any area of the problem domain.

Provided no data dependencies exist, the order of task completion is unimportant. Once all tasks have been computed, the problem is solved. In the preferred bias method the problem domain is divided into equal regions with each region being assigned to a particular processing element, as is done in the balanced data driven approach. However, in this method, these regions are purely *conceptual* in nature. A demand driven model of computation is still used, but the tasks are not now allocated in an arbitrary fashion to the processing elements. Rather, a task is dispatched to a processing element from its conceptual portion. Once all tasks from a processing element's conceptual portion have been completed, only then will that processing element be allocated its next task from the portion of another processing element which has yet to complete its conceptual portion of tasks. Generally this task should be allocated from the portion of the processing element that has completed the least number of tasks. So, for example, from figure 3.19, on completion of the tasks in its own conceptual region, PE_3 may get allocated task number 22 from PE_2's conceptual region. Preferred bias allocation is sometimes also referred to as *conceptual task allocation*.

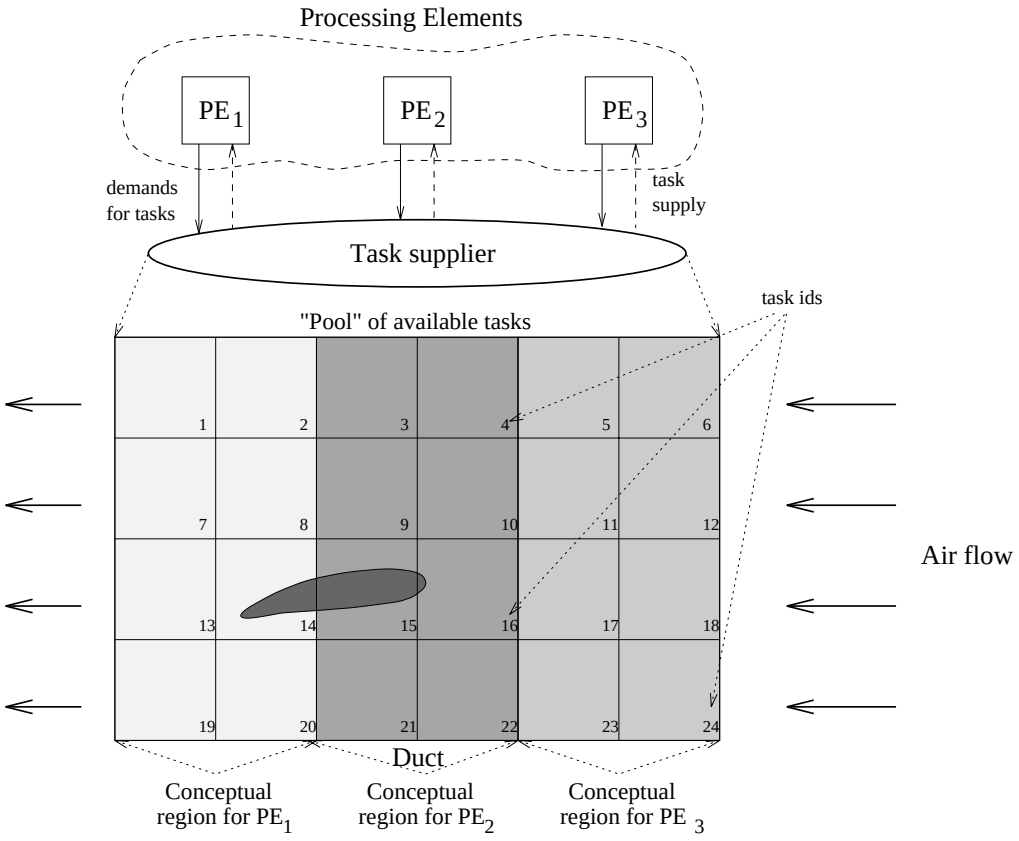


Figure 3.19: Partial result storage balancing by means of conceptual regions

The implications of preferred bias allocation are substantial. The demand driven model's ability to deal with variations in computational complexity is retained, but now the system controller and the processing elements themselves know to whom a task that they have been allocated conceptually belongs. As we will see in section 4.6, this can greatly facilitate the even distribution of partial results at the end of any stage of a multi-stage problem.

The exploitation of data coherence is a vital ploy in reducing idle time due to remote data fetches. Preferred bias allocation of tasks can ensure that tasks from the same region of the problem are allocated to the same processing element. This can greatly improve the cache hit ratio at that processing element.

Chapter 4

Data Management

The data requirements of many problems may be far larger than can be accommodated at any individual processing element. Rather than restricting ourselves to only solving those problems that fit completely within every processing element's local memory, we can make use of the combined memory of all processing elements. The large problem domain can now be distributed across the system and even secondary storage devices if necessary. For this class of application some form of data management will be necessary to ensure that data items are available at the processing elements when required by the computations.

Virtual shared memory regards the whole problem domain as a single unit in which the data items may be individually referenced. This is precisely how the domain could be treated if the problem was implemented on a shared memory multiprocessor system. However, on a distributed memory system, the problem domain is distributed across the system and hence the term **virtual**. Virtual shared memory systems may be implemented at different levels, such as in hardware or at the operating system level. In this chapter we will see how the introduction of a data manager process at each processing element can provide an elegant virtual shared memory at the system software level of our parallel implementation.

4.1 World Model of the Data: No Data Management Required

Not all problems possess very large data domains. If the size of the domain is such that it may be accommodated at every processing element then we say that the processing elements have a "world model" of the data. A world model may also exist if all the tasks allocated to a processing element only ever require a *subset* of the problem domain and this subset can be accommodated completely. In the world model, all principal and additional data items required by an application process will always be available locally at each processing element and thus there is no need for any data item to be fetched from another remote location within the system. If there is no requirement to fetch data items from remote locations as the solution of the problem proceeds then there is no need for any form of data management.

The processor farm described in section 3.4.2 is an example of a parallel implementation which assumes a world model. In this approach, tasks are allocated to processing elements in an arbitrary fashion and thus there is no restriction on which tasks may be computed by which processing element. No provision is made for data management and thus to perform any task, the entire domain must reside at each processing element.

Data items do not always have to be present at the processing element from the start of computation to avoid any form of data management. As discussed in section 3.3.1, both principal and additional data items may be included within a task packet. Provided no further data items are required to complete the tasks specified in the task packet then no data management is required and this situation may also be said to be demonstrating a world data model.

4.2 Virtual Shared Memory

Virtual shared memory provides all processors with the concept of a single memory space. Unlike a traditional shared memory model, this physical memory is distributed amongst the processing elements. Thus, a virtual shared memory environment can be thought of providing each processing element with a *virtual world model* of the problem domain. So, as far as the application process is concerned, there is no difference between requesting a data item that happens to be local, or remote; only the speed of access can be (very) different.

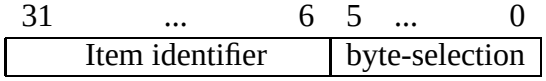
Virtual shared memory can be implemented at any level in the computer hierarchy. Implementations at the hardware level provide a transparent interface to software developers, but requires a specialised machine, such as the DASH system [43]. There have also been implementations at the operating system and compiler level. However, as we shall see, in the absence of dedicated hardware, virtual shared memory can also be easily provided at the system software level. At this level, a great deal of flexibility is available to provide specialised support to minimise any implementation penalties when undertaking the solution of problems with very large data requirements on multiprocessor systems. Figure 4.1 gives four levels at which virtual shared memory (VSM) can be supported, and examples of systems that implement VSM at that particular level.

Higher level	System Software	Provided by the Data Manager process
	Compiler	High Performance Fortran[36], ORCA[6]
	Operating System	Coherent Paging[45]
Lower level	Hardware	DDM [61], DASH [43], KSR-1 [38]

Figure 4.1: The levels where virtual shared memory can be implemented.

4.2.1 Implementing virtual shared memory

At the *hardware level* virtual shared memory intercepts all memory traffic from the processor, and decides which memory accesses are serviced locally, and which memory accesses need to go off-processor. This means that everything above the hardware level (machine code, operating system, etc.) sees a virtual shared memory with which it may interact in exactly the same manner as a physically shared memory. Providing this, so called, transparency to the higher levels, means that the size of data is not determined by the hardware level. However, in hardware, a data item becomes a fixed consecutive number of bytes, typically around 16-256. By choosing the size to be a power of 2, and by aligning data items in the memory, the physical memory address can become the concatenation of the “item-identifier” and the “byte selection”. This strategy is easier to implement in hardware.



In this example, the most significant bits of a memory address locates the data item, and the lower bits address a byte within the item. The choice of using 6 bits as the byte selection in this example is arbitrary.

If a data structure of some higher level language containing two integers of four bytes each happened to be allocated from, say, address ...1100 111100 to ...1101 000100, then item ...1100 will contain the first integer, and item ...1101 will contain the other one. This means that two logically related integers of data are located in two physically separate items (although they could fit in a single data item).

Considered another way, if two unrelated variables, say `x` and `y` are allocated at addresses ...1100 110000 and ...1100 110100, then they reside in the same data item. If they are heavily used on separate processors, this can cause inefficiencies when the machine tries to maintain sequentially consistent copies of `x` and `y` on both processors. The machine cannot put `x` on one processor and `y` on the other, because it does not recognise `x` and `y` as different entities; the machine observes it as a single item that is shared between two processors. If sequential consistency has to be maintained the machine must update every write to `x` and `y` on both processors, even though the variables are not shared at all. This phenomenon is known as *false sharing*.

Virtual shared memory implemented at the *operating system level* also use a fixed size for data items, but these are typically much larger than at the hardware level. By making an item as large as a page of the operating system (around 1-4 KByte), data can be managed at the page level. This is cheaper, but slower than a hardware implementation.

When the *compiler* supports virtual shared memory, a data item can be made exactly as large as any user data structure. In contrast with virtual shared memory implementations at the hardware or operating system level, compiler based implementations can keep logically connected variables together and distribute others. The detection of logically related variables is in the general case very hard, which means that applications written in existing languages such as C, Modula-2 or Fortran cannot be compiled in this way. However, compilers for specially designed languages can provide some assistance. For example, in High Performance Fortran the programmer indicates how arrays should be divided and then the compiler provides the appropriate commands to support data transport and data consistency.

Implementing virtual shared memory at the *system software* level provides the greatest flexibility to the programmer. However, this requires explicit development of system features to support the manipulation of the distributed data item. A *data manager* process is introduced at each processing element especially to undertake this job.

4.3 The Data Manager

Virtual shared memory is provided at the system software level by a data manager process at each processing element. The aim of data management within the parallel system is to ensure the efficient supply of data items to the processing elements. The data manager process manages data items just as the task manager was responsible for maintaining a continuous supply of tasks. Note that the data items being referred to here are the principal and additional data items as specified by the problem domain and not every variable or constant the application process may invoke for the completion of a task.

The application process now no longer deals with the principal and additional data items directly, but rather indirectly using the facilities of the data manager. The application process achieves this by issuing a data request to the data manager process every time a data item is required. The data manager process assumes the responsibility for ensuring that every request for a data item from the application process will be satisfied. The data manager attempts to satisfy these requests by maintaining a local data cache.

The data management strategies implemented by the data manager and outlined in the following sections are *active*, dynamically requesting and acquiring data items during computation. This means that an application process should *always* have its request for a data item satisfied immediately by the data manager unless:

- at the start of the problem the application processes make requests before any initial data items have been provided by the system controller;
- the data manager's data fetch strategy has failed in some way.

4.3.1 The local data cache

The concept of *data sharing* may be used to cope with very large data requirements [14, 23]. Data sharing implements virtual shared memory by allocating every data item in the problem domain an unique identifier. This allows a required item to be "located" from somewhere within the system, or from secondary storage if necessary. The size of problem that can now be tackled is, therefore, no longer dictated by the size of the local memory at each processing element, but rather only by the limitations of the combined memory plus the secondary storage.

The principal data item required by an application process is specified by the task it is currently performing. Any additional data item requirements are determined by the task *and* by the algorithm chosen to solve the problem. These additional data items may be known *a priori* by the nature of the problem, or they may only become apparent as the computation of the task proceeds.

To avoid any processing element idle time, it is essential that the data manager has the required data item available locally at the moment the application process issues a request for it. In an attempt to achieve this, the data manager maintains a local cache of data items as shown in figure 4.2. The size of this cache, and thus the number of data items it can contain, is determined by the size of a processing element's local memory.

Each data item in the system is a packet containing the unique identifier, shown in figure 4.2 as `id`, together with the actual data which makes up the item. The data items may be permanently located at a specific processing element, or they may be free to migrate within the system to where they are required. When a data manager requires a particular data item which is not already available locally, this data item must be fetched from some remote location and placed into the local cache. This must occur before the application process can access the data item. The virtual shared memory of the system is thus the combination of the local caches at all the processing elements plus the secondary storage which is under the control of the file manager at the system controller.

In certain circumstances, as will be seen in the following sections, rather than removing the data item from the local cache in which it was found, it may be sufficient simply to take a copy of the data item and return this to the local cache. This is certainly the case when the data items within the problem domain are *read-only*, that is the values of the data items are not altered during the course of the parallel solution of the problem (and

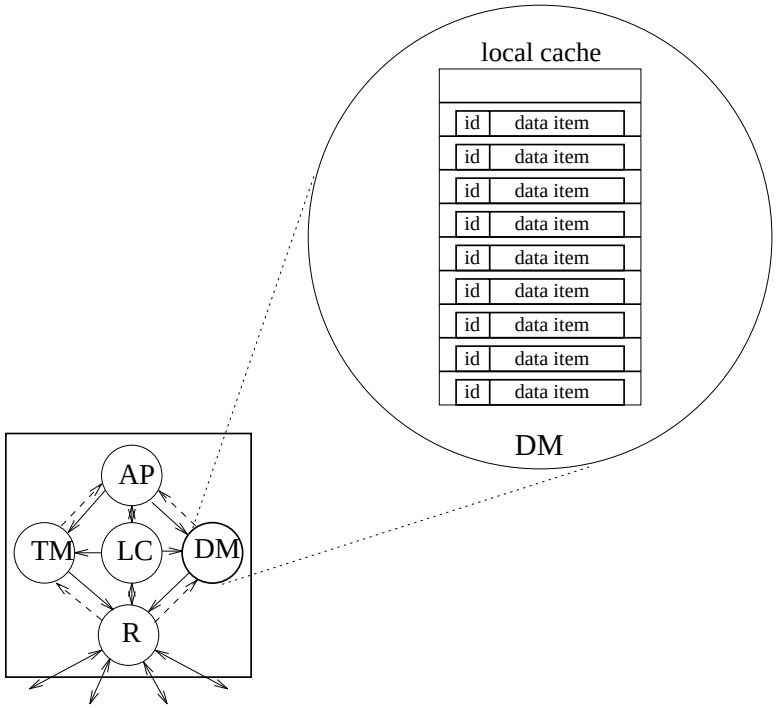


Figure 4.2: The local cache at the data manager

indeed the same would be true of the sequential implementation). This means that it is possible for copies of the same data item to be present in a number of local caches. Note that it is no advantage to have more than one copy of any data item in one local cache.

There is a limited amount of space in any local cache. When the cache is full and another data item is acquired from a remote location, then one of the existing data items in the local cache must be replaced by this new data item. Care must be taken to ensure that no data item is inadvertently completely removed from the system by being replaced in all local caches. If this does happen then, assuming the data item is read-only, a copy of the entire problem domain will reside on secondary storage, from where the data items were initially loaded into the local caches of the parallel system. This means that should a data item being destroyed within the system, another copy can be retrieved from the file manager (FM) of the system controller.

If the data items are *read-write* then their values may be altered as the computation progresses. In this case, the data managers have to beware of consistency issues when procuring a data item. The implications of consistency will be discussed in section 4.4.

As we will now see, the strategies adopted in the parallel implementation for acquiring data items and storing them in the local caches can have a significant effect on minimising the implementation penalties and thus improving overall system performance. The onus is on the data manager process to ensure these strategies are carried out efficiently.

4.3.2 Requesting data items

The algorithm being executed at the application process will determine the next data item required. If the data items were all held by the application process, requesting the data

item would be implemented within the application process as an “assignment statement”. For example a request for data item `i` would simply be written as `x := data_item[i]`. When all the data items are held instead by the data manager process, this “assignment statement” must be replaced by a request from the application process to the data manager for the data item followed by the sending of a copy of the data item from the local cache of the data manager to the waiting application process, as shown in figure 4.3.

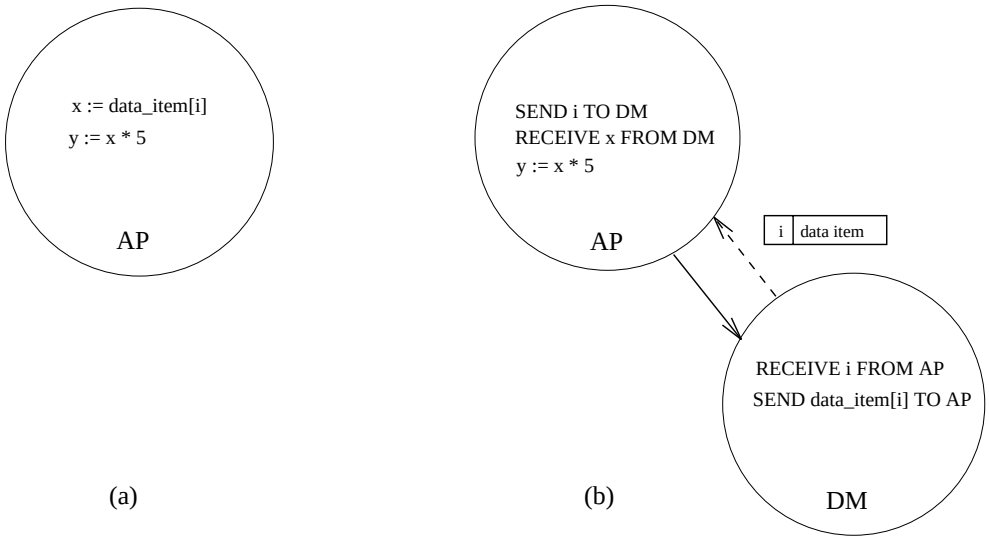


Figure 4.3: Accessing a data item (a) with, and (b) without a data manager

The data item’s unique identifier enables the data manager to extract the appropriate item from its local cache. If a data item requested by the application process is available, it is immediately transferred, as shown in figure 4.4(a). The only slight delay in the computation of the application process will occur by the need to schedule the concurrent data manager and for this process to send the data item from its local cache. However, if the data item is not available locally then the data manager must “locate” this item from elsewhere in the system. This will entail sending a message via the router to find the data item in another processing element’s local cache, or from the file manager of the system controller. Having been found, the appropriate item is returned to the requesting data manager’s own local cache and then finally a copy of the item is transferred to the application process.

If the communicated request from the application process is asynchronous and this process is able to continue with its task while awaiting the data item then no idle time occurs. However, if the communication with the data manager is synchronous, or if the data item is essential for the continuation of the task then idle time will persist until the data item can be fetched from the remote location and a copy given to the application process, as shown in figure 4.4(b). Unless otherwise stated, we will assume for the rest of this chapter that an application process is unable to continue with its current task until its data item request has been satisfied by the data manager.

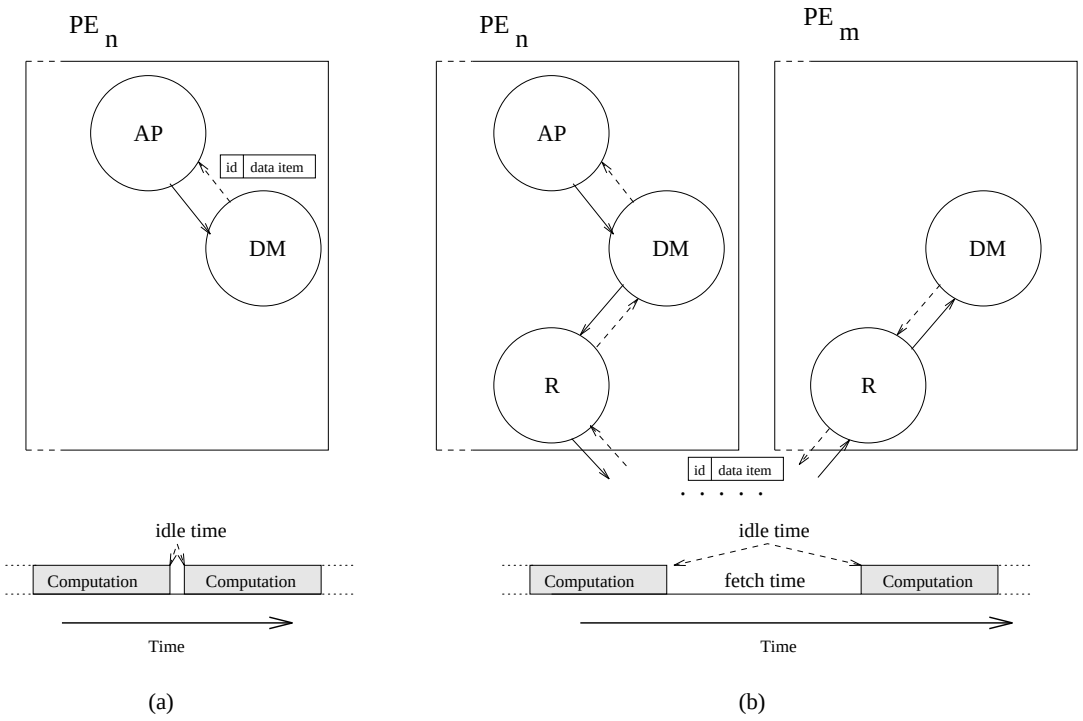


Figure 4.4: AP idle time due to: (a) Data item found locally (b) Remote data item fetch

4.3.3 Locating data items

When confronted with having to acquire a remote data item, two possibilities exist for the data manager. Either it knows exactly the location of the data item within the system, or this location is unknown and some form of search will have to be instigated.

Resident sets

Knowing the precise location of the requested data item within the system enables the data manager to instruct the router to send the request for the data item, directly to the appropriate processing element.

One of the simplest strategies for allocating data items to each processing element's local cache is to divide all the data items of the problem domain evenly amongst the processing elements before the computation commences. Providing there is sufficient local memory and assuming there are n processing elements, this means that each processing element would be allocated $\frac{1}{n}^{th}$ of the total number of data items. If there isn't enough memory at each processing element for even this fraction of the total problem domain then as many as possible could be allocated to the local caches and the remainder of the data items would be held at the file manager of the system controller. Such a simplistic scheme has its advantages. Provided these data items remain at their predetermined local cache for the duration of the computation, then the processing element from which any data item may found can be computed directly from the identity of the data item.

For example, assume there are twelve data items, given the unique identification numbers $1, \ldots, 12$, and three processing elements, PE_1, PE_2, and PE_3. A predetermined al-

location strategy may allocate permanently data items $1, \ldots, 4$ to PE_1, data items $5, \ldots, 8$ to PE_2 and $9, \ldots, 12$ to PE_3. Should PE_2 wish to acquire a copy of data item 10, it may do so directly from the processing element known to have that data item, in this case PE_3.

It is essential for this simple predetermined allocation strategy that the data items are not overwritten or moved from the local cache to which they are assigned initially. However, it may be necessary for a processing element to also acquire copies of other data items as the computation proceeds, as we saw with PE_2 above. This implies that the local cache should be partitioned into two distinct regions:

- a region containing data items which may never be replaced, known as the *resident set*; and,
- a region for data items which may be replaced during the parallel computation.

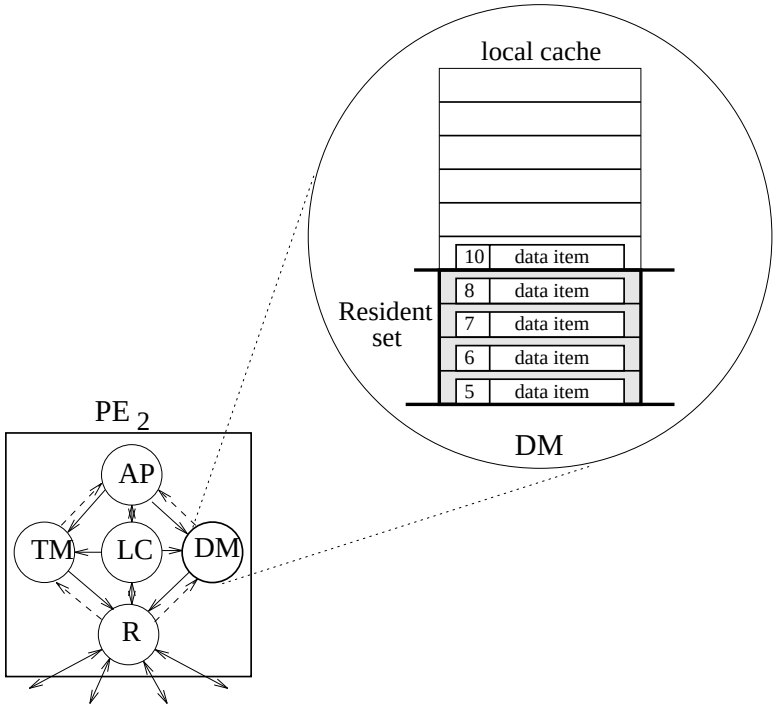


Figure 4.5: Resident set of the local cache

The size of the resident set should be sufficient to accommodate all the pre-allocated data items, as shown for PE_2 from the above example in figure 4.5. The remaining portion of the local cache will be as large as allowed by the local memory of the processing element. Note that this portion needs to have sufficient space to hold a minimum of *one* data item as this is the maximum that the application process can require at any specific point during a task's computation. To complete a task an application process may require many data items. Each of these data items may in turn replace the previously acquired one in the single available space in the local cache.

The balanced data driven model of computation is well suited to a simple pre-determined even data item allocation scheme. In this model the system controller knows prior to the computation commencing precisely which tasks are to be assigned to which processing elements. The same number of tasks is assigned to each processing element and thus the

principal data items for each of these tasks may be pre-allocated evenly amongst the local caches of the appropriate processing elements. Similar knowledge is available to the system controller for the unbalanced data driven model, but in this case the number of tasks allocated to each processing element is not the same and so different numbers of principal data items will be loaded into each resident set. Note that the algorithm used to solve the problem may be such that, even if a data driven model is used and thus the principal data items are known in advance, the additional data items may not be known *a priori*. In this case, these additional data items will have to be fetched into the local caches by the data managers as the computation proceeds and the data requirements become known.

More sophisticated pre-allocation strategies, for example some form of hashing function, are possible to provide resident sets at each processing element. It is also not necessary for each data item to be resident at only one processing element. Should space permit, the same data item may be resident at several local caches.

The pre-allocation of resident sets allows the location of a data item to be determined from its unique identifier. A pre-allocated resident set may occupy a significant portion of a local cache and leave little space for other data items which have not been pre-allocated. The shortage of space would require these other data items to be replaced constantly as the computation proceeds. It is quite possible that one data item may be needed often by the same application process either for the same task or for several tasks. If this data item is not in the resident set for that processing element, then there is the danger that the data item will be replaced during the periods that it is not required and thus will have to be re-fetched when it is required once more. Furthermore, despite being pre-allocated, the data items of a resident set may in fact never be required by the processing element to which they were allocated. In the example given earlier, PE_2 has a resident set containing data items $5, \ldots, 8$. Unless there is *a priori* knowledge about the data requirements of the tasks, there is no guarantee that PE_2 will ever require any of these data items from its resident set. In this case, a portion of PE_2's valuable local cache is being used to store data items which are never required, thus reducing the available storage for data items which are needed. Those processing elements that do require data items $5, \ldots, 8$ are going to have to fetch them from PE_2. Not only will the fetches of these data items imply communication delays for the requesting data managers, but also, the need for PE_2's data manager to service these requests will imply concurrent activity by its data manager which will detract from the computation of the application process.

The solution to this dilemma is not to pre-allocate resident sets, but to build up such a set as computation proceeds and information is gained by each data manager as to the data items most frequently used by its processing element. Profiling can also assist in establishing these resident sets, as explained in section 4.5.3. The price to pay for this flexibility is that it may no longer be possible for a data manager to determine precisely where a particular data item may be found within the system.

Searching for data at unknown locations

Acquiring a specific data item from an unknown location will necessitate the data manager requesting the router process to "search" the system for this item. The naive approach would be for the router to send the request to the data manager process of each processing element in turn. If the requested data manager has the necessary data item it will return a copy and then there is no need for the router to request any further processing elements. If the requested data manager does not have the data item then it must send

back a `not_found` message to the router, whereupon the next processing element may be tried. The advantages of this *one-to-one* scheme is that as soon as the required data item is found, no further requests need be issued and only one copy of the data item will ever be returned. However, the communication implications of such a scheme for a large parallel system are substantial. If by some quirk of fate (or Murphy's law), the last processing element to be asked is the one which has the necessary data item, then one request will have resulted in $2 \times (number\ of\ PEs\ - 1)$ messages, a quite unacceptable number for large systems. Furthermore, the delay before the data item is finally found will be large, resulting in long application process idle time.

An alternative to this communication intensive *one-to-one* approach, is for the router process to issue a global broadcast of the request; a *one-to-many* method. A bus used to connect the processing elements is particularly suited to such a communication strategy, although, as discussed in section 2.2.1, a bus is not an appropriate interconnection method for large multiprocessor systems. The broadcast strategy may also be used efficiently on a more suitable interconnection method for large systems, such as interconnections between individual processors. In this case, the router issues the request to its directly-connected neighbouring processing elements. If the data managers at these processing elements have the required data item then it is returned, if not then these neighbouring processing elements in turn propagate the request to their neighbours (excluding the one from which they received the message). In this way, the requests propagates through the system like ripples on a pond. The message density inherent in this approach is significantly less than the *one-to-one* approach, however one disadvantage is that if the requested data item is replicated at several local caches, then several copies of the same data item will be returned to the requesting data manager, when only one is required.

For very large multiprocessor systems, even this *one-to-many* approach to discovering the unknown location of a data item may be too costly in terms of communication latency and its contribution to message density within the system. A compromise of the direct access capabilities of the pre-allocated resident set approach and the flexibility of the dynamic composition of the local caches is the notion of a *directory* of data item locations.

In this approach, it is not necessary to maintain a particular data item at a fixed processing element. We can introduce the notion of a *home*-processing element that knows where that data item is, while the data item is currently located at the *owner*-processing element. The home-processing element is fixed and its address may be determined from the identifier of the data item. The home processing element knows which processing element is currently owning the data item. Should this data item be subsequently moved and the one at the owner-process removed, then either the home-processing element must be informed as to the new location of the data item or the previous owner-processing element must now maintain a pointer to this new location. The first scheme has the advantage that a request message may be forwarded directly from the home-processing element to the current owner, while the second strategy may be necessary, at least for a while after the data item has been moved from an owner, to cope with any requests forwarded by the home-processing element before it has received the latest location update.

Finally, it is also possible to do away with the notion of a home-processing element, by adding a hierarchy of directories. Each directory on a processing element "knows" which data items are present on the processing element. If the required data item is not present, a directory higher up in the hierarchy might know if it is somewhere nearby. If that directory does not know, yet another directory might if it is further away. This is much like the organisation of libraries: you first check the local library for a book, if they

do not have it you ask the central library, and so on until you finally query the national library. With this organisation there is always a directory that knows the whereabout of the data item, but it is very likely that the location of the data item will be found long before asking the highest directory. (The Data Diffusion Machine [61] and KSR-1 [38] used a similar strategy implemented in hardware).

4.4 Consistency

Copies of *read-only* data items may exist in numerous local caches within the system without any need to "keep track" of where all the copies are. However, if copies of *read-write* data items exist then, in a virtual shared memory system, there is the danger that the data items may become *inconsistent*. The example in figure 4.6 illustrates this problem of inconsistency. Suppose that we have two processing elements PE_1 and PE_2, and a data item y with a value 0, that is located at processing element PE_1. Processing Element PE_2 needs y, so it requests and gets a copy of y. The data manager on processing element PE_2 decides to keep this copy for possible future reference. When the application at processing element PE_1 updates the value of y, for example by overwriting it with the value 1, processing element PE_2 will have a *stale* copy of y. This situation is called *inconsistent*: if the application running at processing element PE_1 requests y it will get the new value (1), while the application at processing element PE_2 will still read the old value of y (0). This situation will exist until the data manager at processing element PE_2 decides to evict y from its local memory.

The programming model of a physical shared memory system maintains only one copy of any data item; the copy in the shared memory. Because there is only one copy, the data items cannot become inconsistent. Hence, naive *virtual* shared memory differs from *physical* shared memory in that virtual shared memory can become inconsistent.

To maintain consistency all copies of the data items will have to be "tracked down" at certain times during the parallel computation. Once again tone-to-one or many-to-one methods could be used to determine the unknown locations of copies of the data items. If the directory approach is used then it will be necessary to maintain a complete "linked list" through all copies of any data item, where each copy knows where the next copy is, or it knows that there are no more copies. A consistency operation is performed on this list by sending a message to the first copy on the list, which then ripples through the list. These operations thus take a time linear in the number of copies. This is expensive if there are many copies, so it can be more efficient to use a tree structure (where the operation needs logarithmic time). (A combination of a software and hardware tree directory of this form is used in the LimitLESS directory [12].)

There are several ways to deal with this inconsistency problem. We will discuss three options: data items are kept consistent at all times (known as sequential consistency); the actual problem somehow copes with the inconsistencies (known as weak consistency); and finally, inconsistent data items are allowed to live for a well defined period (the particular scheme discussed here is known as release consistency).

4.4.1 Keeping the data items consistent

The first option is that the data manager will keep the data items consistent at all times. To see how the data items can be kept consistent, observe first that there are two conditions that must be met before a data item can become inconsistent. Firstly, the data item must be

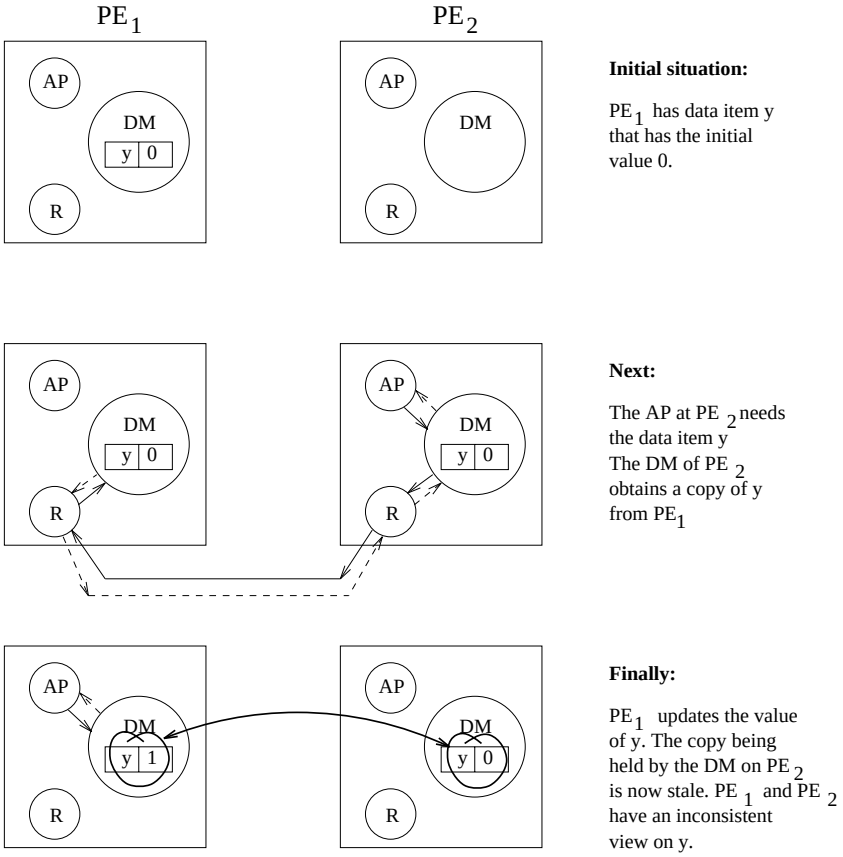


Figure 4.6: An example how an inconsistency arises. There are two processing elements, PE_1 and PE_2 and a data item y. PE_2 keeps a copy of y, while y is updated at PE_1.

duplicated; as long as there is only a single copy of the data item, it cannot be inconsistent. Secondly, some processing element must update one of the copies, without updating the other copies. This observation leads to two protocols that the data manager can observe to enforce consistency, while still allowing copies to be made:

1. Ensure that there is not more than a single copy of the data item when it is updated. This means that before a write all but one of the copies must be deleted. This solution is known as an *invalidating protocol*.

2. Ensure that all copies of the data item are replaced when it is updated. This solution is known as an *updating protocol*.

It is relatively straightforward to check that the invalidating option will always work: all copies are always identical, because a write only occurs when there is only a single copy. In the example, the copy of y at processing element PE_2 will be destroyed before y is updated on processing element PE_1.

For the updating protocol to be correct, the protocol must ensure that all copies are replaced "at the same time". Suppose that this is not the case: in the example the value on processing element PE_1 might be updated, while processing element PE_2 still has an old value for y. If the data managers running on processing elements PE_1 and PE_2 communicate, they can find out about this inconsistency. In order for the update protocol to work, the updating data manager must either ensure that no other data manager is accessing the data item while it is being updated, or that it is impossible for any communication (or other update) to overtake this update.

It is not easy to decide in general whether an invalidating or an updating protocol is better. Below are two examples that show that invalidating and updating protocols both have advantages and disadvantages. In both cases we assume that the problem is running on a large number of processing elements, and that there is a single shared data item that is initially replicated over all processing elements.

1. A task, or tasks, being performed by an application process at one processing element might require that the data item be updated at this data manager many times, without any of the other processing elements using it. An updating protocol will update all copies on all processing elements during every update, even though the copies are not being used on any of the other processing elements.

   An invalidating protocol is more efficient, because it will invalidate all outstanding copies once, whereupon the application process can continue updating the data item without extra communication.

2. Suppose that instead of ignoring the data item, all other processing elements do need the updated value. An invalidating protocol will invalidate all copies and update the data item, whereupon all other processing elements have to fetch the value again. This fetch is on demand, which means that they will have to wait on the data item.

   An updating protocol does a better job since it distributes the new value, avoiding the need for the other processing elements to wait for it.

There is a case for (and against) both protocols. It is for this reason that these two protocols are sometimes combined. This gives a protocol that, for example, invalidates all copies that have not been used since the last update, and updates the copies that were used since the last update. Although these hybrid protocols are potentially more efficient, they are unfortunately often more complex than a pure invalidating or updating protocol.

4.4.2 Weak consistency: repair consistency on request

The option to maintain sequential consistency is an expensive one. In general, an application process is allowed to proceed with its computation only after the invalidate or update has been completed. In the example of the invalidating protocol, all outstanding copies must have been erased and the local copy must have been updated before the application process can proceed. This idle time may be an unacceptable overhead. One of the ways to reduce this overhead is to forget about maintaining consistency automatically. Instead, the local cache will stay inconsistent until the application process orders the data manager to repair the inconsistency.

There are two important advantages of weak consistency. Firstly, the local cache is made consistent at certain points in the task execution only, reducing the overhead. Secondly, local caches can be made consistent in parallel. Recall for example, the updating protocol of the previous section. In a weakly consistent system we can envisage that every write to a data item is asynchronously broadcasted to all remote copies. Asynchronously means that the processing element performing the write continues whether the update has been completed or not. Only when a consistency-command is executed must the application process wait until all outstanding updates are completed. In the same way, a weakly consistent invalidating protocol can invalidate remote copies in parallel. These optimisations lead to further performance improvement. The disadvantage of weak consistency is the need for the explicit commands within the algorithm at each application process so that when a task is being executed, at the appropriate point, the data manager can be instructed to make the local cache consistent.

4.4.3 Repair consistency on synchronisation: Release consistency

A weak consistency model as sketched above requires the programmer of the algorithm to ensure consistency at any moment in time. Release consistency is based on the observation that algorithms do not go from one phase to the other without first synchronising. So it suffices to make the local caches consistent during the synchronisation operation. This means that immediately after each synchronisation the local caches are guaranteed to be consistent. This is in general slightly more often than strictly necessary, but it is far less often than would be the case when using sequential consistency. More importantly, the application process itself does not have to make the local caches consistent anymore, it is done "invisibly".

Note that although invisible, consistency is only restored during an explicit synchronisation operation; release consistency behaves still very differently from sequential consistency. As an example, an application process at PE_1 can poll a data item in a loop, waiting for the data item to be changed by the application process at PE_2. Under sequential consistency any update to the data item will be propagated, and cause the application process at PE_1 to exit the loop. Under release consistency updates do not need to be propagated until a synchronisation point, and because it does not recognise that the polling loop is actually a synchronisation point the application process at PE_1 might be looping forever.

4.5 Minimising the Impact of Remote Data Requests

Failure to find a required data item locally means that the data manager has to acquire this data item from elsewhere within the system. The time to fetch this data item and,

therefore, the application process idle time, can be significant. This *latency* is difficult to predict and may not be repeatable due to other factors, such as current message densities within the system. The overall aim of data management is to maximise effective processing element computation by minimising the occurrence and effects of remote data fetches. A number of techniques may be used to reduce this latency by:

Hiding the Latency: - overlapping the communication with the computation, by:

Prefetching - anticipating data items that will be required

Multi-threading - keeping the processing element busy with other useful computation during the remote fetch

Minimising the Latency: - reducing the time associated with a remote fetch by:

Caching & profiling - exploiting any coherence that may exist in the problem domain

4.5.1 Prefetching

If it is known at the start of the computation which data items will be required by each task then these data items can be *prefetched* by the data manager so that they are available locally when required. The data manager thus issues the requests for the data items *before* they are actually required and in this way overlaps the communication required for the remote fetches with the ongoing computation of the application process. This is in contrast with the simple fetch-upon-demand strategy where the data manager only issues the external request for a data item at the moment it is requested by the application process and it is not found in the local cache.

By treating its local cache as a "circular buffer" the data manager can be loading prefetched data items into one end of the buffer while the application process is requesting the data items from the other end, as shown in figure 4.7. The "speed" at which the data manager can prefetch the data items will be determined by the size of the local cache and the rate at which the application process is "using" the data items.

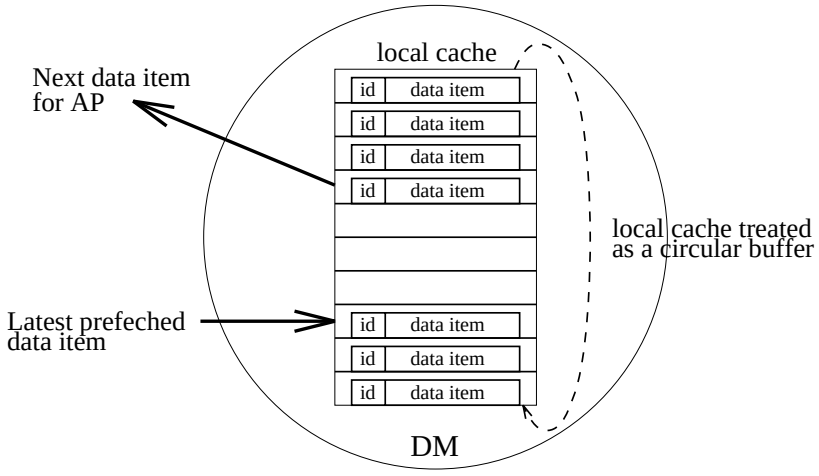


Figure 4.7: Storing the prefetched data items in the local cache

This knowledge about the data items may be known *a priori* by the nature of problem. For example, in a parallel solution of the hemi-cube radiosity method, the data manager knows that each task, that is the computation of a single row of the matrix of form factors, requires all the environment's patch data. The order in which these data items are considered is unimportant, as long as all data items are considered. The data manager can thus continually prefetch those data items which have yet to be considered by the current task. Note that in this problem, because all the data items are required by every task and the order is unimportant (we are assuming that the local cache is not sufficiently big to hold all these data items), those data items which remain in the local cache at the end of one task are also required by the subsequent task. Thus, at the start of the next task, the first data item in the local cache can be forwarded to the application process and prefetching can commence once more as soon as this has happened.

The choice of computation model adopted can also provide the information required by the data manager in order to prefetch. The principal data items for both the balanced and unbalanced data driven models will be known by the system controller before the computation commences. Giving this information to the data manager will enable it to prefetch these data items. A prefetch strategy can also be used for principal data items within the preferred bias task allocation strategy for the demand driven computation model, as described in section 3.4.5. Knowledge of its processing element's conceptual region can be exploited by the data manager to prefetch the principal data items within this region of the problem domain.

4.5.2 Multi-threading

Any failure by the data manager to have the requested data item available locally for the application process will result in idle time unless the processing element can be kept busy doing some other useful computation.

One possibility is for the application process to save the current state of a task and commence a new task whenever a requested data item is not available locally. When the requested data item is finally forthcoming either this new task could be suspended and the original task resumed, or processing of the new task could be continued until it is completed. This new task may be suspended awaiting a data fetch and so the original task may be resumed. Saving the state of a task may require a large amount of memory and indeed, several states may need to be saved before one requested data item finally arrives. Should the nature of the problem allow these stored tasks to in turn be considered as task packets, then this method has the additional advantage that these task packets could potentially be completed by another processing element in the course of load balancing, as explained in the section 3.4.4 on distributed task management.

Another possible option is multi-threading. In this method there is not only one, but several application processes on each processing element controlled by an application process controller (APC), as shown in figure 4.8. Each application process is known as a separate *thread* of computation. Now, although one thread may be suspended awaiting a remote data item, the other threads may still be able to continue. It may not be feasible to determine just how many of these application processes will be necessary to avoid the case where all of them are suspended awaiting data. However, if there are sufficient *threads* (and of course sufficient tasks) then the processing element should always be performing useful computation. Note that multi-threading is similar to the Bulk Synchronous Parallel paradigm [60].

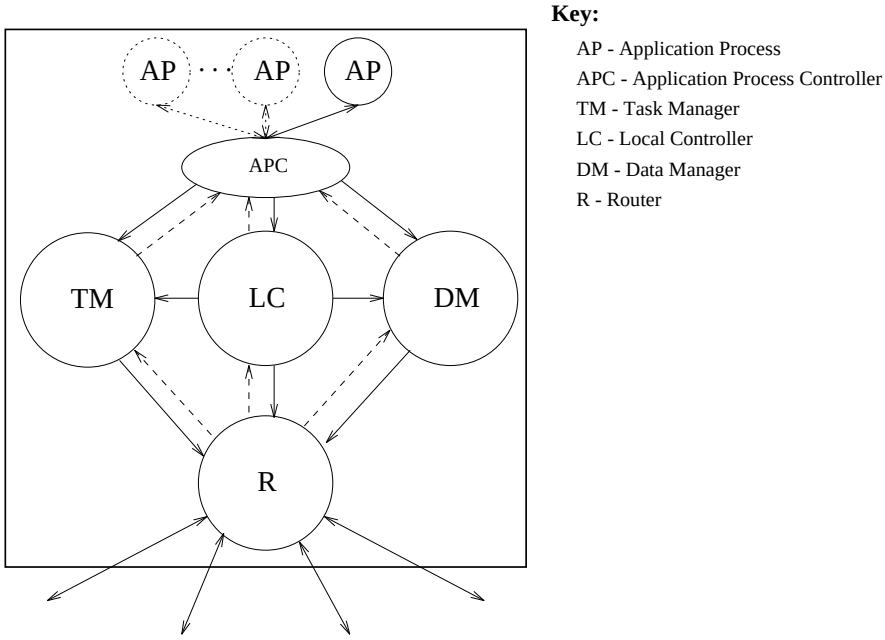


Figure 4.8: Several application processes per processing element

One disadvantage of this approach is the overhead incurred by the additional context switching between all the application processes and the application process controller, as well as the other system software processes: the router, task manager and the data manager, that are all resident on the same processor. A variation of multiple active threads is to have several application processes existing on each processing element, but to only have *one* of them active at any time and have the application process controller manage the scheduling of these processes explicitly from information provided by the data manager. When an application processes' data item request cannot be satisfied locally, that process will remain descheduled until the data item is forthcoming. The data manager is thus in a position to inform the application process controller to activate another application process, and only reactivate the original application process once the required data has been obtained. Note the application process controller schedules a new application process by sending it a task to perform. Having made its initial demand for a task to the application process controller (and not the task manager as discussed in section 3.4.2) an application process will remain descheduled until explicitly rescheduled by the application process controller.

Both forms of multi-threading have other limitations. The first of these is the extra memory requirements each thread places on the processing elements local memory. The more memory that each thread will require, for local constants and variables etc, the less memory there will be available for the local cache and thus fewer data items will be able to be kept locally by the data manager. A "catch 22" (or is that "cache 22") situation now arises as fewer local data items implies more remote data fetches and thus the possible need for yet more threads to hide this increase in latency. The second difficulty of a large number of threads running on the same processing element is the unacceptably heavy overhead that may be placed on the data manager when maintaining the local cache. For

example, a dilemma may exist as to whether a recently fetched data item for one thread should be overwritten before it has been used if its "slot" in the local cache is required by the currently active thread.

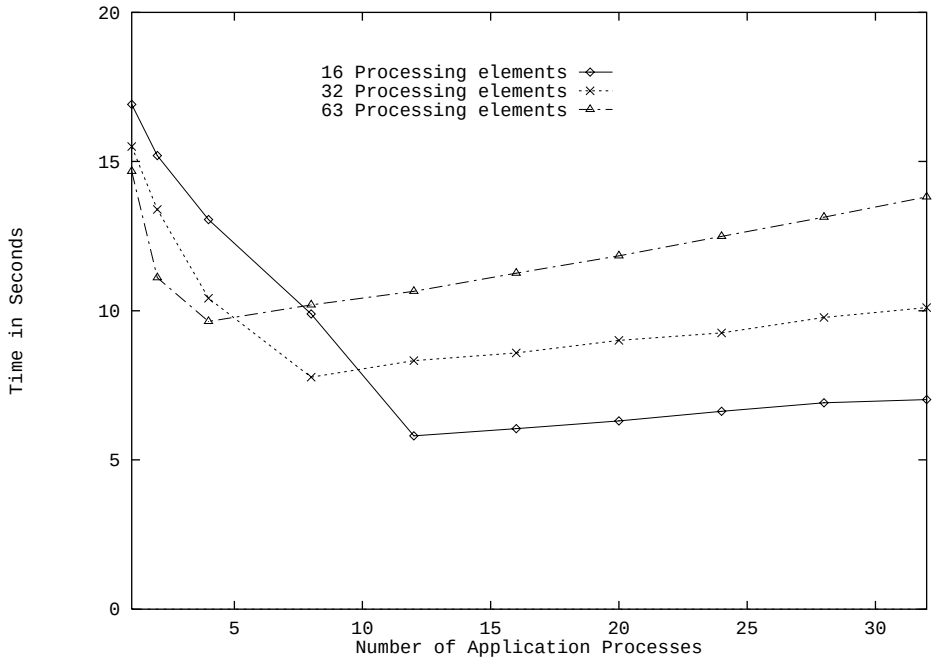


Figure 4.9: Problem solution time in seconds

Figure 4.9 shows results for a multi-threaded application. The graph shows the time in seconds to solve a complex parallel ray tracing problem with large data requirements using more than one application process per processing element. As can be seen, increasing the number of application processes per processing element produces a performance improvement until a certain number of threads have been added. Beyond this point, the overheads of having the additional threads are greater than the benefit gained, and thus the times to solve the problem once more increase. The number of threads at which the overheads outweigh the benefits gained is lower for larger numbers of processing elements. This is because the more application processes there are per processing element, the larger the message output from each processing element will be (assuming an average number of remote fetches per thread). As the average distances the remote data fetches have to travel in larger systems is greater, the impact of increasing numbers of messages on the overall system density is more significant and thus the request latency will be higher. Adding more threads now no longer helps overcome communication delays, but in fact, the increasing number of messages actually exacerbates the communication difficulties. Ways must be found of dynamically scheduling the optimum number of application processes at each processing element depending on the current system message densities.

Despite these shortcomings, multi-threading does work well, especially for low numbers of threads and is a useful technique for avoiding idle time in the face of unpredictable data item requirements. Remember that multiple threads are only needed at a processing element if a prefetch strategy is not possible and the data item required by one thread was not available locally. If ways can be found to try and guess which data items are likely to

required next then, if the data manager is right at least some of the time, the number of remote fetches-on-demand will be reduced. Caching and profiling assist the data manager with these predictions.

4.5.3 Profiling

Although primarily a task management technique, profiling is used explicitly to assist with data management, and so is discussed here. At the start of the solution of many problems, no knowledge exists as to the data requirements of any of the tasks. (If this knowledge did exist then a prefetching strategy would be applicable). Monitoring the solution of a single task provides a list of all the data items required by that task. If the same monitoring action is carried for all tasks then at the completion of the problem, a complete "picture" of the data requirements of all tasks would be known. Profiling attempts to predict the data requirements of future tasks from the list of data requirements of completed tasks.

Any spatial coherence in the problem domain will provide the profiling technique with a good estimate of the future data requirements of those tasks from a similar region of the problem domain. The data manager can now use this profiling information to *prefetch* those data items which are *likely* to be used by subsequent tasks being performed at that processing element. If the data manager is always correct with its prediction then profiling provides an equivalent situation to prefetching in which the application process is never delayed awaiting a remote fetch. Note in this case there is no need for multi-threading.

A simple example of spatial coherence of the problem domain is in shown in figure 4.10. This figure is derived from figure 3.3 which showed how the principal data item (PDI) and additional data items (ADIs) made up a task. In figure 4.10 we can see that task i and task j come from the same region of the problem domain and spatial coherence of the problem domain has meant that these two tasks have three additional data items in common. Task k, on the other hand, is from a different region of the problem domain, requires only one additional data item which is not common to either task i or task j.

Thus, the more successful the predictions are from the profiling information, the higher will be the cache-hit ratios. From figure 4.10 on page 83 we can see that if the completion of task i was used to profile the data item requirements for task j then, thanks to the spatial coherence of task i to task j in the problem domain, the data manager would have a 66% success rate for the additional data items for task j. However, a similar prediction for the additional data items for task k would have a 0% success rate and result in a 100% cache-miss, that is all the additional data items for task k would have to be fetched-on-demand.

4.6 Data Management for Multi-Stage Problems

In section 3.3.3 we discussed the algorithmic and data dependencies that can arise in problems which exhibit more than one distinct stage. In such problems, the results from one stage become the principal data items for the subsequent stage as was shown in figure 3.12. So, in addition to ensuring the application processes are kept supplied with data items during one stage, the data manager also needs to be aware as to how the partial results from one stage of the computation are stored at each processing element in anticipation of the following stage.

This balancing of partial result storage could be achieved statically by all the results of a stage being returned to the system controller. At the end of that current stage the system controller is in a position to distribute this data evenly as the principal and additional data

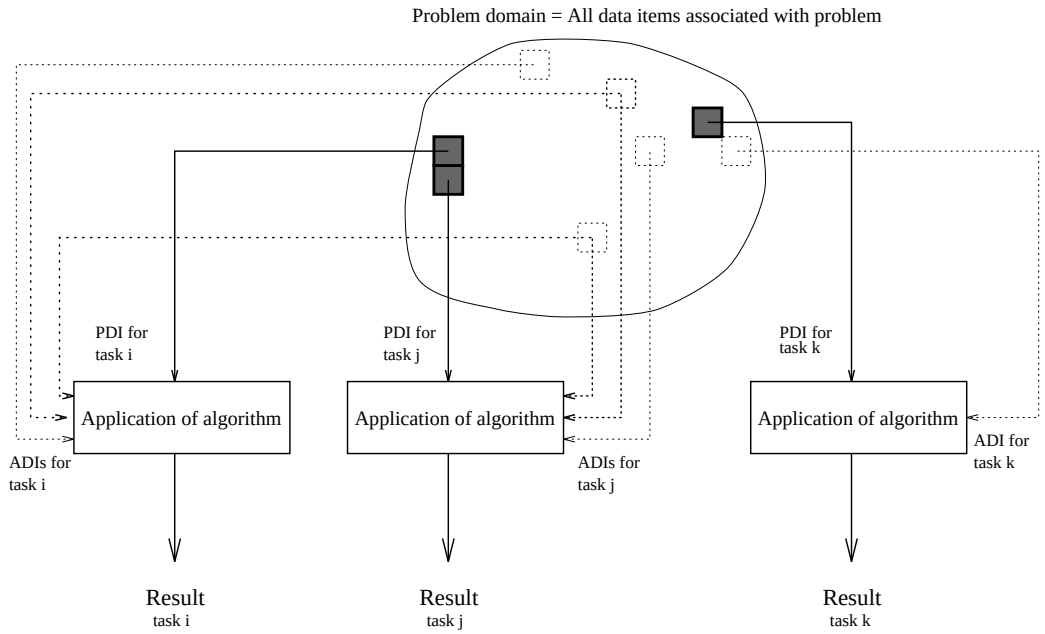


Figure 4.10: Common additional data items due to spatial coherence of the problem domain

items for the next stage of the problem. The communication of these potentially large data packets twice, once during the previous stage to the system controller and again from the system controller to specific processing elements, obviously may impose an enormous communication overhead. A better static distribution strategy might be to leave the results in place at the processing elements for the duration of the stage and then have them distributed from the processing elements in a manner prescribed by the system controller. Note that in such as scheme the local cache of each processing element must be able to hold not only the principal and additional data items for the current stage, but also have space in which to store these partial results in anticipation of the forthcoming stage. It is important that these partial results are kept separate so that they are not inadvertently overwritten by data items during the current stage.

In a demand driven model of computation the uneven computational complexity may result in a few processing elements completing many more tasks than others. This produces a flaw in the second static storage strategy. The individual processing elements may simply not have sufficient space in their local cache to store more than their fair share of the partial results until the end of the stage.

Two dynamic methods of balancing this partial result data may also be considered. Adoption of the preferred bias task management strategy, as discussed in section 3.4.5, can greatly facilitate the correct distribution of any partial results. Any results produced by one processing element from another's conceptual portion, due to task load balancing, may be sent to directly to this other processing element. The initial conceptual allocation of tasks ensures that the destination processing element will have sufficient storage for the partial result.

If this conceptual allocation is not possible, or not desirable, then balancing the partial results dynamically requires each processing element to be kept informed of the progress of all other processing elements. This may be achieved by each processing element broad-

casting a short message on completion of every task to all other processing elements. To ensure that this information is as up to date as possible, it is advisable that these messages have a special high priority so that they may be handled immediately by the router processes, by-passing the normal queue of messages. Once a data manager's local cache reaches its capacity the results from the next task are sent in the direction of the processing element that is known to have completed the least number of tasks and, therefore, the one which will have the most available space. To further reduce the possible time that this data packet may exist in the system, any processing element on its path which has storage capacity available may absorb the packet and thus not route it further.

Bibliography

[1] W. B. Ackerman. Data flow languages. In N. Gehani and A. D. McGettrick, editors, *Concurrent Programming*, chapter 3, pages 163–181. Addison-Wesley, 1988.

[2] G. S. Almasi and A. Gottleib. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, California, 2^{nd} edition, 1994.

[3] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS*, volume 30, Atlantic City, Apr. 1967. AFIPS Press, Reston, Va.

[4] M. Annaratone et al. Warp architecture and implementation. In *13^{th} Annual International Symposium on Computer Architecture*, pages 346–356, Tokyo, June 1986.

[5] J. Backus. Can programming be liberated from the von Neumann style functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[6] H. Bal. *Programming Distributed Systems*. Silicon Press, Summit, New Jersey, 1990.

[7] A. Basu. A classification of parallel processing systems. In *ICCD*, 1984.

[8] K. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, 29(9):836–840, Sept. 1980.

[9] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, Wokingham, England, 1990.

[10] A. W. Burks. Programming and structural changes in parallel computers. In W. Händler, editor, *Conpar*, pages 1–24, Berlin, 1981. Springer.

[11] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Massachusetts, 1990.

[12] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4^{th} International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV*, pages 224–234, Apr. 1991.

[13] A. G. Chalmers. Occam - the language for educating future parallel programmers? *Microprocessing and Microprogramming*, 24:757–760, 1988.

[14] A. G. Chalmers and D. J. Paddon. Communication efficient MIMD configurations. In *4^{th} SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, 1989.

[15] P. Chaudhuri. *Parallel Algorithms: Design and analysis*. Prentice-Hall, Australia, 1992.

[16] B. Codenotti and M. Leonici. *Introduction to parallel processing*. Addison-Wesley, Wokingham, England, 1993.

[17] A. L. DeCegama. *The Technology of Parallel Processing: Parallel Processing Architectures and VLSI Design*. Prentice-Hall International Inc., 1989.

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

[19] V. Faber, O. M. Lubeck, and A. B. White Jr. Super-linear speedup of an efficient sequential algorithm is not possible. *Parallel Computing*, 3:259–260, 1986.

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

[21] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[22] C. F. Gerald and P. O. Wheatley. *Applied numerical analysis*. World Student Series. Addison-Wesley, Reading, MA, 5^{th} edition, 1994.

[23] S. A. Green and D. J. Paddon. A non-shared memory multiprocessor architecture for large database problems. In M. Cosnard, M. H. Barton, and M. Vanneschi, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, Pisa, 1988.

[24] H. A. Grosch. High speed arithmetic: The digital computer as a research tool. *Journal of the Optical Society of America*, 43(4):306–310, Apr. 1953.

[25] H. A. Grosch. Grosch's law revisited. *Computerworld*, 8(16):24, Apr. 1975.

[26] J. L. Gustafson. Re-evaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[27] D. R. Hartree. The ENIAC, an electronic computing machine. *Nature*, 158:500–506, 1946.

[28] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[29] T. Hey. Scientific applications. In G. Harp, editor, *Transputer Applications*, chapter 8, pages 170–203. Pitman Publishing, 1989.

[30] D. W. Hillis. *The Connection Machine*. The MIT Press, 1985.

[31] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*, chapter 1, pages 60–81. Adam Hilger, 1988.

[32] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, 1988.

[33] M. Homewood, M. D. May, D. Shepherd, and R. Shepherd. The IMS T800 transputer. *IEEE Micro*, pages 10–26, 1987.

[34] R. M. Hord. *Parallel Supercomputing in MIMD Architectures*. CRC Press, Boca Raton, 1993.

[35] R. J. Hosking, D. C. Joyce, and J. C. Turner. *First steps in numerical analysis*. Hodder and Stoughton, Lonfon, 1978.

[36] HPF Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1), June 1993.

[37] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill Series in Computer Science. McGraw-Hill, New York, 1993.

[38] KSR. *KSR Technical Summary*. Kendall Square Research, Waltham, MA, 1992.

[39] V. Kumar, A. Grama, A. Gupta, and G. Karyps. *Introduction to Parallel Computing*. Benjamin/Cummings, Redwood City, California, 1994.

[40] H. T. Kung. *VLSI array processors*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[41] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In Duff and Stewart, editors, *Sparse Matrix proceedings*, Philadelphia, 1978. SIAM.

[42] C. Lazou. *Supercomputers and Their Use*. Claredon Press, Oxford, revised edition, 1988.

[43] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan. 1993.

[44] T. Lewis and H. El-Rewini. *Introduction to parallel computing*. Prentice-Hall, 1992.

[45] K. Li. Ivy: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing*, 2:94–101, Aug. 1988.

[46] G. J. Lipovski and M. Malek. *Parallel Computing: Theory and comparisons*. John Wiley, New York, 1987.

[47] M. D. May and R. Shepherd. Communicating process computers. Inmos technical note 22, Inmos Ltd., Bristol, 1987.

[48] L. F. Menabrea and A. Augusta(translator). Sketch of the Analytical Engine invented by Charles Babbage. In P. Morrison and E. Morrison, editors, *Charles Babbage and his Calculating Engines*. Dover Publications, 1961.

[49] D. Nussbaum and A. Argarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, Mar. 1991.

[50] B. Purvis. Programming the Intel i860. *Parallelogram International*, pages 6–9, Oct. 1990.

[51] M. J. Quinn. *Parallel Computing: Theory and practice*. McGraw-Hill, New York, 1994.

[52] V. Rajaraman. *Elements of parallel computing*. Prentice-Hall of India, New Dehli, 1990.

[53] S. F. Reddaway. DAP - a Distributed Array Processor. In *1^{st} Annual Symposium on Computer Architecture*, 1973.

[54] R. M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21:63–72, 1978.

[55] J. E. Shore. Second thoughts on parallel processing. *Comput. Elect. Eng.*, 1:95–109, 1973.

[56] R. J. Swam, S. H. Fuller, and D. P. Siewiorek. 'Cm^*—A Modular, Multi-Microprocessor'. In *Proc. AFIPS 1977 Fall Joint Computer Conference 46*, pages 637–644, 1977.

[57] S. Thakkar, P. Gifford, and G. Fiellamd. The Balance multiprocessor system. *IEEE Micro*, 8(1):57–69, Feb. 1988.

[58] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data driven and demand-driven computer architecture. *Communications of the ACM*, 14(1):95–143, Mar. 1982.

[59] A. Trew and G. Wilson, editors. *Past, Present and Parallel: A survey of available parallel computer systems*. Springer-Verlag, London, 1991.

[60] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[61] D. H. D. Warren and S. Haridi. The Data Diffusion Machine—a scalable shared virtual memory multiprocessor. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 943–952, Tokyo, Japan, Dec. 1988.

[62] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

Chapter 5

Classification of Parallel Rendering Systems

In the preceding chapters, we have reviewed the fundamental concepts of parallel processing and given some indication of how it might be effectively used in graphics rendering. Since many types of parallel rendering have been investigated [Gree91] [Whit92], classifying the various schemes is important to characterize the behavior of each. A parallel rendering system can be classified according to the method of task subdivision and/or by the hardware used to implement the scheme. Often, the choice of one influences the other.

Classifying by task subdivision refers to the method in which the original rendering task is broken into smaller pieces to be processed in parallel. Obviously, such subdivision strongly depends on the type of rendering employed. A task for rendering polygons will offer a different set of subdivision opportunities than a ray tracing task. Also included in these decisions is the type of load balancing technique to employ.

Ultimately the rendering scheme is implemented within some sort of parallel environment. The system may run on parallel hardware (e.g., a general multiprocessor or specialized hardware) or in a distributed computing environment (a group of individual machines working together to solve a single problem). The advantages and disadvantages associated with each environment are discussed below.

5.1 Classification by Task Subdivision

In this section, we will look at two different types of rendering (polygon-based rendering and ray tracing) and various methods for subdividing the original task into subtasks for parallel processing. Although many subdivision techniques exist for each, we will focus on the schemes most widely used. For each technique, recall that our goal is to subdivide the original task in such a way as to maximize parallelism, while not creating excessive overhead.

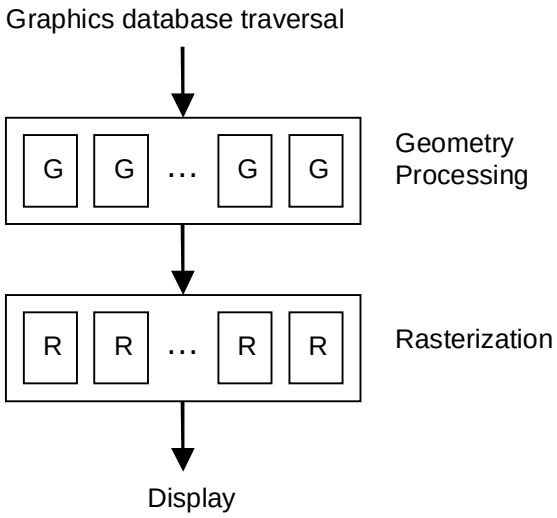


Figure 5.1 Polygon rendering pipeline

5.1.1 Polygon Rendering

For polygon rendering, we often deal with a very large number of primitives (e.g., triangles) which can often be processed in a parallel manner. To handle this type of rendering, a graphics pipeline is usually employed (see figure 5.1). Stages in this pipeline include geometry processing and rasterization.

Geometry processing comprises transformation, clipping, lighting, and other tasks associated with a primitive. A straightforward method for parallelizing geometry processing is to assign each processor a subset of primitives (or objects) in the scene to render. In rasterization, scan-conversion, shading, and visibility determination are performed. To parallelize this processing, each processor could perform the pixel calculations for a small part of the final image.

One way to view the processing of primitives is as a problem of sorting primitives to the screen since a graphics primitive can fall anywhere on or off the screen [Moln94]. For a parallel system, we need to distribute data across processors to keep the load balanced. Actually, this sort can occur anywhere in the rendering pipeline:

- during geometry processing (sort-first)
- between geometry processing and rasterization (sort-middle)
- during rasterization (sort-last)

The structure of the parallel rendering system is determined by the location of this sort. The following discussion follows that in [Moln94].

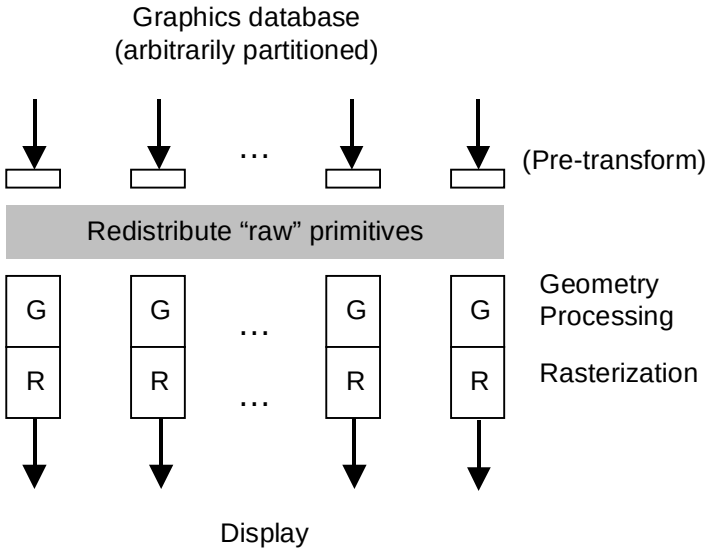


Figure 5.2 Sort-first polygon rendering scheme

5.1.1.1 Sort-First

The main idea behind sort-first is to distribute primitives early (during geometry processing) in the rendering pipeline (see figure 5.2). The screen is divided into regions of equal size (see figure 5.3), and each processor (or renderer) is assigned a region. Each processor is responsible for all the pixel calculations that are associated with its screen region.

In an actual implementation, primitives would initially be assigned to processors in an arbitrary way. Each renderer then performs enough transformation processing to determine the screen region into which the primitive falls. If this region belongs to another processor, the primitive is sent over the interconnection network to that processor for rendering. After each primitive has been placed with the proper renderer, all of the processors can work in parallel to complete the final image.

With this method, each processor implements the entire rendering pipeline for its portion of the screen. Communication costs can be kept comparatively low compared with other methods if features such as frame coherence are properly exploited. For rendering a single frame, however, almost all the primitives will have to be redistributed after the initial random assignment. Some duplication of effort may occur if a primitive falls into more than one region, or if the results of the original geometry processing are not sent with the primitive when it is transmitted to the appropriate renderer. Also, the system is susceptible to load imbalance since primitives may be concentrated into particular regions or may simply take longer to render. Both of these situations

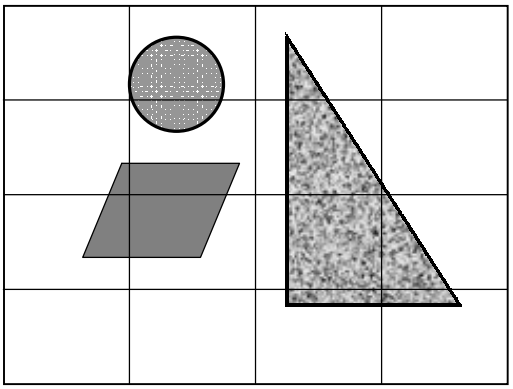


Figure 5.3 Image space subidivision

will cause the affected processor to consume more time in processing its screen region. Very few, if any, sort-first renderers have been built.

5.1.1.2 Sort-Middle

In a sort-middle renderer, primitives are sorted and redistributed in the middle of the pipeline: between geometry processing and rasterization (see figure 5.4). By this point, the screen coordinates of the primitives have been determined through transformation processing, but the

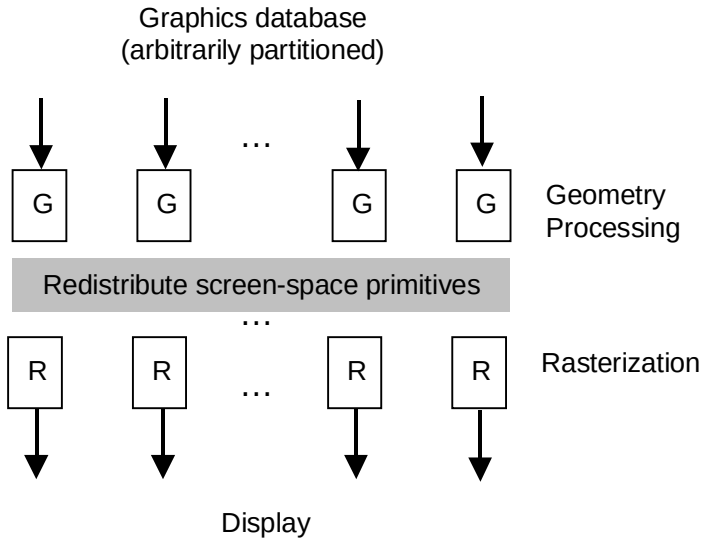


Figure 5.4 Sort-middle polygon rendering scheme

primitives have not yet been rasterized. This point is a natural breaking position in the rendering pipeline.

In an actual implementation, primitives are arbitrarily assigned to processors as before. The geometry processors perform transformation, lighting, and other processing on the primitives originally assigned to them, and then classify the primitives according to screen region. Screen-space primitives are then sent to the appropriate rasterizing processors, which have been assigned unique regions of the screen, for the remaining processing.

The sort-middle strategy is general and straightforward and has been implemented in both hardware (including Pixel-Planes 5 [Fuch89] and the SGI Reality Engine [Akel93]) and software [Whit94] [Ells94]. Like sort-first, however, this method is susceptible to load imbalance due to the uneven distribution of primitives across screen space. Communication times can be higher under certain conditions. Also, primitives that overlap regions may require some additional processing.

5.1.1.3 Sort-Last

Under the sort-last strategy, sorting is deferred until the end of the rendering pipeline (see figure 5.5). Primitives are rasterized into pixels, samples, or pixel fragments, which are then transmitted to the appropriate processor for visibility determination.

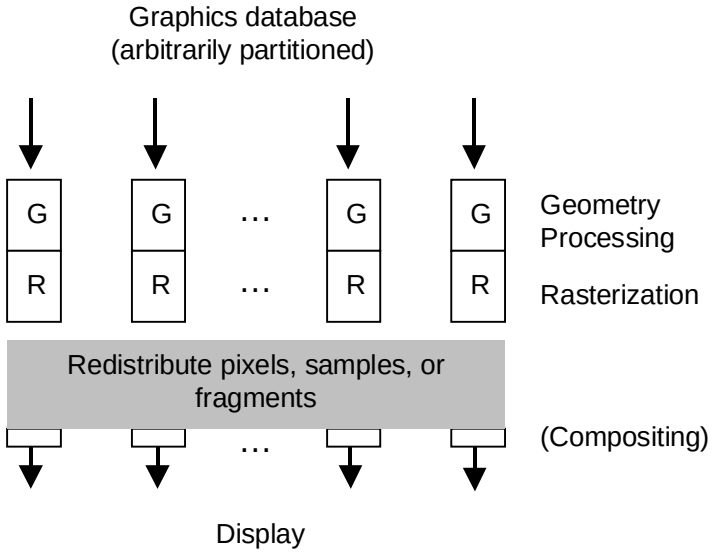


Figure 5.5 Sort-last polygon rendering scheme

In practice, primitives are initially distributed to processors in an arbitrary manner, as in the other methods. Each renderer then performs the operations necessary to compute the pixel values for its primitives, regardless of where those pixels may reside on the screen. These values are then sent to the appropriate processors according to screen location. At this point, the rasterizing processors perform visibility calculations and composite the pixels for final display.

As in sort-first, each processor implements the entire graphics pipeline for its primitives. While the overall technique is less prone to load imbalance, the pixel traffic in the final sort can be very high. Numerous rendering systems using the sort-last method have been constructed in various forms, including [Evan92] and [Kubo93].

5.1.2 Ray Tracing

Ray tracing is a powerful rendering technique that can produce high-quality graphics images; however, this quality comes at a price of intensive calculation and long rendering times. Even relatively simple ray-traced animations can prohibitively expensive to render on a single processor. For longer, more complex animations, the rendering time can be intractable. Fortunately, ray tracing is a prime candidate for parallelization since its processing is readily amenable to subdivision. Specifically, ray tracing inherently contains a large amount of parallelism due to the independent nature of its pixel calculations [Whit80]; therefore, most ray tracing rendering algorithms lend themselves to parallelization in screen space.

Other partitioning schemes are employed in ray tracing as well. Instead of dividing the image space, the object space can be split into smaller regions, or the objects themselves may be assigned to individual processors. These techniques are discussed more fully below.

5.1.2.1 Image Space Partitioning

Using this scheme, the viewing plane is divided into regions, each of which is completely rendered by an individual processor (see figure 5.3). That is, for each pixel in a region, the processor assigned to that region computes its entire ray tree. While this technique is conceptually straightforward, the entire database of scene objects must be accessible to every processor.

The benefits of this approach are simplicity and low interprocessor communication as compared with other partitioning methods; the largest drawback is its limitation to multiprocessor architectures with significant local processor memory. Another potential problem is load imbalance, since image detail may be concentrated in certain regions of the screen. To combat

this situation, the load balancing algorithm may further subdivide complex regions to provide idle processors with additional tasks.

5.1.2.2 Object Space Partitioning

Here the 3-dimensional space where the scene objects reside is divided into subvolumes, or voxels. Voxels may not be equally sized in order to achieve better load balancing. In the initialization phase of ray tracing, each voxel is parceled out to a particular processor. When rays are cast during rendering, they are passed from processor to processor as they travel through the object space. Each processor, therefore, needs only the scene information associated with its assigned voxels.

While this technique may not suffer from frequent load imbalance, it does incur costs in other ways. First, as new rays are shot, they must tracked through voxel space; this processing is not required for other schemes. Additionally, since potentially millions of rays are fired for each image, communication could be become excessive as rays enter and exit regions of object space during rendering.

5.1.2.3 Object Partitioning

This partitioning scheme parallelizes the rendering task by assigning each object to an individual processor. As in object space partitioning, rays are passed as messages between processors, which in turn test the ray for intersection with the objects they are assigned. Object partitioning also shares some of the benefits and detriments of object space partitioning. Specifically, the load may be fairly well-balanced, but the communication costs may be high due to the large amount of ray message traffic.

5.1.2.4 Load Balancing Scheme

Finally, parallel rendering schemes can be classified according to their load balancing method. Of course, the primary goal of any load balancing scheme is to distribute the work among processors as evenly as possible and thus exploit the highest degree of parallelism available in the application. Many different types of load balancing schemes exist, but each falls into one of two categories:

- *Static Load Balancing.* In this scheme, partitioning is performed up front and processors are assigned subtasks for the entire duration of the rendering process. In this way, overhead is minimized later in the rendering; however, a good deal of care must be taken

to ensure that the load will be balanced. Otherwise, the algorithm will suffer from poor performance.

- *Dynamic Load Balancing.* With this scheme, some processing assignments are determined at the start, but later assignments are demand-driven. That is, when a processor determines that it needs more work to do, it will request a new assignment. In this way, processors will never be idle as long as more work is left to do. The key here is to distribute the load as evenly as possible without incurring excessive overhead.

5.1.2.5 Comments

Hybrid schemes have also been proposed, combining image space partitioning with object space partitioning [Bado94] and image space partitioning with object partitioning [Kim96]. When choosing a partitioning scheme, the architecture of the parallel machine should be considered. For instance, an image space partitioning algorithm will perform better on a MIMD machine than on a SIMD machine. In general, tradeoffs exist between the type of partitioning algorithm used and the architecture chosen.

5.2 Classification by Hardware

As previously stated, for some computationally intensive rendering tasks, parallel processing provides the only practical means to a solution. One way to perform parallel rendering is to use a single multiprocessor machine, such as a Thinking Machines CM-5, Intel Paragon, Cray T3E, or specialized parallel processor. In these machines, enormous computing power is provided by up to tens of thousands of processing elements able to access many gigabytes of memory and to work in concert through a high-speed interconnection network. Multiprocessors are the most powerful computers in the world and play an active role in solving Grand Challenge problems, such as weather prediction, fluid dynamics, and drug design [Hwan93].

An alternative to using traditional multiprocessor systems for parallel processing is to employ a network of workstations acting as a single machine. This approach, termed distributed or cluster computing, is conceptually similar to a multiprocessor, but each processing element consists of an independent machine connected to a network usually much slower than a multiprocessor interconnection network. While this network can be of any type (e.g., Ethernet, ATM) or topology, the computers connected to it are generally UNIX-based machines which support some type of distributed programming environment, such as Parallel Virtual Machine (PVM) [Geis94] or Message Passing Interface (MPI) [Grop94]. Many types of applications can benefit from distributed computing, including computation-intensive graphics tasks, such as ray tracing [Sung96].

This section focuses on past work that has been documented using traditional multiprocessors and clusters of machines to accomplish graphics rendering tasks, particularly in the area of ray tracing. Included in this discussion is relevant background concerning PVM and MPI, as well as motivation for using these systems in a clustered environment.

5.2.1 Parallel Hardware

Since rendering consumes such a large amount of computing resources and time, a good deal of effort has gone into exploring parallel solutions on multiprocessor machines. Some of the schemes proposed are designed to run on general-purpose parallel machines, such as the CM-5, while others rely on specialized hardware built especially for ray tracing rendering. A brief survey of these techniques appears in the following sections. Although current research continues in the design and implementation of parallel rendering systems, a flurry of activity in this area occurred in the late1980s and early 1990s, as reflected in many of the references.

5.2.1.1 General-Purpose Multiprocessors

In [Plun85], a vectorized ray tracer is proposed for the CDC Cyber 205. In a given execution cycle, rays awaiting processing are distributed to individual processors and ray-object calculations are performed object by object in a lock-step SIMD fashion.

Similarly, [Crow88] implements a SIMD ray tracing algorithm, but for the Connection Machine (CM-2). Image subdivision is used with one pixel being assigned to each of the 16K processors to produce a 128x128 image, with ray-object intersections performed on an object by object basis. The algorithm proposed by [Schr92] also runs on a CM-2, but uses an object-space subdivision coupled with processor remapping capabilities to achieve dynamic load balancing.

Rounding out the SIMD field, [Goel96] describes a ray casting method developed on the MasPar MP-1 for volume rendering, another computation-intensive graphics application used for viewing complex structures in medical imaging and other forms of scientific visualization. To handle the large amount of data and processing involved, machines are assigned portions of the volume to render, which are composited to produce a final image. This system allows users to rotate a volume, magnify areas of interest, and perform other viewing operations.

In the MIMD category, [Reis97] employs an IBM SP/2 running an image-space partitioning scheme with dynamically adjustable boundaries to render frames of an animation progressively. In this form of rendering, termed progressive rendering, an image is initially rendered quickly at low resolution and progressively refined when little or no user interaction takes place. Progressive rendering is useful in interactive environments where frame generation rate is important. The goal

of [Keat95] also involves progressive rendering, although their renderer makes use of object-space partitioning on the Kendall Square Research KSR1 machine.

Several research efforts have focused on the Intel iPSC machines as the architectural environment for implementing a parallel ray tracer. Interestingly, whether the partitioning scheme is image-based [Isle91] [Silv94], object space-based [Prio88] [Prio89] or a hybrid of the two [Akti94] [Bado94], the load balancing scheme is almost always of a static nature ([Isle91] also tests a dynamic scheme). This choice results from a concern that dynamic load balancing schemes produce a large number of messages, which in turn, may dramatically affect the performance of a distributed machine [Prio89].

In the area of transputer-based machines, [Gree90] uses an image subdivision technique combined with memory and cache local to each processor to deal with the many required accesses to the scene description database. Here, the granularity of parallelism is controlled through the size of the image subregion, which also relates directly to the effectiveness of the dynamic load balancing scheme. To render ray-traced animations, [Maur93] use a static object-space partitioning scheme on a system of 36 transputers. Progressive ray tracing and volume rendering on transputers is addressed by [Sous90] and [Pito93], respectively.

5.2.1.2 Specialized Multiprocessors

Probably one of the most noteworthy examples of a specialized multiprocessor for polygon rendering is the series of Pixel-Planes machines developed at UNC-Chapel Hill. The Pixel-Planes 4 machine [Eyle88] is a SIMD machine with three basic components: a host workstation, a graphics processor, and a frame buffer. Each of the customized processors is responsible for a column of display pixels. For its time, it provided good performance; however, the system used processors with slow clock speeds and did not provide effective load balancing.

Pixel-Planes 5 [Fuch89] provided some improvements over Pixel-Planes 4 by incorporating faster processors and employing a more flexible MIMD architecture. The system implemented a sort-middle algorithm, with each processor in charge of a particular region of the screen. To handle the communication, a ring architecture capable of handling eight messages simultaneously is employed. Ultimately, the ring network imposes a limit on scalability.

The PixelFlow machine [Moln92] was developed to overcome the limitations of the previous architectures through parallel image composition. Each individual processor works to create a full-screen image using only the primitives assigned to it. All of these images are collected and composited to form the final display.

For ray tracing, [Lin91] employs a specialized SIMD machine to perform stochastic ray tracing. The stochastic method adds extra processing to the ray tracing algorithm to handle antialiasing, an important aspect of any renderer. To overcome some of the inefficiencies found in

other SIMD approaches, a combination of image space partitioning and object space partitioning is used. That is, a block of pixels is rendered by casting rays and using scene coherence to restrict the parts of object space which must be tested.

In [Gaud88] a special-purpose MIMD architecture using image space subdivision and a static load distribution is described. To overcome the problem of having the entire object database resident at each processor, a central broadcast processor issues data packets describing the object database cyclically. Here, the processors make requests for various pieces of the database, and only those parts are broadcast in a given cycle. Using a somewhat different approach, [Shen95] uses object space partitioning on clusters of processors, but each processor operates in a pipelined fashion, a scheme previously explored in the LINKS-1 architecture [Nish83].

One of the few multiprocessor architectures which allocates work based on object subdivision combined with image subdivision is proposed by [Kim96]. Each processor handles ray-object intersection tests with its assigned objects, which are spread across the object space. If the load becomes unbalanced, objects are dynamically transferred to other processors.

Other specialized multiprocessor machines of note are the Pixel Machine [Potm89] (useful for several types of rendering including ray tracing and the RayCasting Engine [Meno94] (specifically built for CSG modeling).

5.2.1.3 Distributed Computing Environments

Parallel rendering using distributed computing environments continues to grow in popularity, especially in the fields of entertainment and scientific visualization. Below are a few interesting examples.

Perhaps the most popular example is the Disney film, *Toy Story*, which used a network of 117 Sun workstations and the Pixar Renderman system to produce the animation [Henn96]. To generate its 144,000 individual frames, *Toy Story* required about 43 years of CPU time. If not for the many machines participating in the computation, the movie's production could not be realized. For some tasks, such as applying surface textures, one machine was chosen as a server for the rest. For other tasks, such as final rendering, the machines were basically used independently to render individual frames.

Another entertainment application used a network of 40 Amiga machines to render special effects for the television series *SeaQuest* [Worl93]. Although the delivered product contained only two to three minutes of computer graphics per episode (3,600 to 5,400 frames of animation), the rendering activity was so time-intensive that the team struggled to deliver the graphics within its weekly deadline.

For the average user, some popular commercial animation packages (e.g., Alias/Wavefront,

Maya, and 3D Studio) employ coarse-grain parallelism to allow rendering of individual frames of an animation across a network of machines. This technique can mean the difference between an animation being ready in hours or in days. POV-Ray has also been ported to run in a clustered environment; however, the parallelization scheme works on single images only.

Other computation-intensive graphics problems have taken advantage of the processing power of distributed computing, specifically volume rendering and virtual reality. Although real-time interaction in these systems is constrained by the relatively slow network connecting the machines, significant speedups have been reported using a network of IBM RS/6000 machines for volume rendering [Gier93] [Ma93], and a network of Sun Sparcstations and HP workstations for virtual environments [Pan96].

Distributed computing is also being applied to computer vision algorithms [Judd94], a field closely related to graphics. Here, researchers use PVM on a cluster of 25 Sun Sparcstations for an edge-detection algorithm. One remarkable result of their experiments is that they achieved superlinear speedup on the cluster over the sequential version. This result is due to the large aggregate memory of the clustered machines, which reduced the amount of paging as compared to the single processor.

Not nearly as much research has been conducted concerning ray tracing in distributed computing environments as in traditional multiprocessor machines. Perhaps this fact is due to the relatively recent introduction of PVM and MPI. Regardless of the reason, more advanced parallel ray tracing algorithms combined with a distributed computing environment remain a largely unexplored area. Several related projects are summarized below.

For single images, [Jeva89] uses a dynamic load balancing technique with spatial partitioning and a novel warp synchronization method. At the other end of the spectrum, [Ris94] applies a static load balancing scheme using object partitioning on a network composed of both sequential workstations and parallel computers. Surprisingly few systems use an image partitioning scheme in a distributed computing environment, even though it represents the technique with the highest potential for speedup [Clea86] and overcomes the problems of limited local memory that exist in traditional multiprocessors.

For animations, [DeMa92] describes the DESIRe (Distributed Environment System for Integrated Rendering) system, which incorporates a coarse-level dynamic load balancing scheme that distributes individual frames of an animation to networked workstations. The goal of [Stob88] is similar, except that the system is designed to run without affecting the regular users of the workstations. By stealing idle cycles from 22-34 workstations, a ray-traced animation lasting five minutes (7550 frames) was rendered in two months, although the overall task was estimated at 32 CPU-months.

The work presented in [Cros95] uses a relatively small (three-machine) distributed environment for ray tracing animations in virtual reality applications. Here, each of the machines has a special task assigned to it according to its processing specialty. In order to achieve close to

interactive rates, the system, which takes advantage of progressive refinement, is composed of fairly powerful individual processors connected by an ATM network.

5.2.2 Discussion of Architectural Environments

For parallel processing tasks, the fastest systems will generally be the specialized multiprocessor machines, since they are built with a specific task in mind. Next will be general-purpose multiprocessor machines. Although distributed environments may provide the same number of processors as a multiprocessor machine, computations will be performed more quickly on multiprocessors due to their high-speed interconnection networks. Even so, several factors have motivated a trend toward distributed computing.

First, and perhaps most importantly, not many organizations can afford a parallel machine, which can easily cost millions of dollars [Geis94]. Many sites, however, already have some type of network of computers. Second, multiprocessors often employ specialized or exotic hardware and software resources that significantly increase the complexity, and hence the cost, of the machine; conversely, great expense is rarely incurred to perform distributed computing because the network and the machines are usually already in place. Surprisingly, distributed computing has proven to be so cost-effective that networks of standard workstations have been purchased specifically to run parallel applications that were previously executed on more expensive supercomputers [Grop94].

Due to the fact that networks of workstations are loosely coupled, distributed computing environments allow the network to grow in stages and take advantage of the latest network technology. As CPUs evolve to faster speeds, workstations can be swapped out for the latest model. Such flexibility in network and processor choice is not usually available on a multiprocessor. Another consideration is system software: operating system interface, editors, compilers, debuggers, etc. A benefit of workstation platforms is that they remain relatively stable over time, allowing programmers to work in familiar environments. To use multiprocessor systems, developers may have to climb a steep learning curve.

Additionally, in a distributed computing networked environment, the interconnected computers often consist of a wide variety of architectures and capabilities. This heterogeneity leads to a rich variety of machine combinations and computing possibilities, which can be tailored to specific applications to reduce overall execution time. On the other hand, a multiprocessor machine does not spend processing time converting data between various machine types, as a distributed computing environment might.

Finally, while utilization and efficiency are extremely important in the multiprocessor world, users on a network of traditional machines rarely consider these issues. The results are under-utilized computers which spend much of their time idle. With distributed computing, some of those idle cycles can be put to good use without impacting the primary users of the machines.

5.2.3 Message-Passing Software for Distributed Computing Environments

To realize distributed computing, computers in a network must support some type of distributed programming environment that allows users to write parallel applications for networked machines. This programming environment should provide a common interface for developers to pass messages easily across various network types and between machines of differing architectures. Although many additional features are usually included, a distributed programming environment need only provide a minimum set of capabilities to be useful [Grop94]:

- First, some method must exist to start up and initialize the parallel processes on all participating machines. This procedure may be as simple as specifying each machine and an associated command in a static file, or spawning the processes directly within the program of the master process. Here, the master process refers to a user-initiated process responsible for delegating work and compositing results; conversely, slave processes perform only the work assigned to them and report results back to the master.

- Once start-up is complete, a process should be able to identify itself, as well as other processes running on the local machine or remote machines participating in the work. Such identification is useful for specifying the source and destination of transmitted messages.

- Since a distributed computing environment often consists of machines with widely varying architectures, message transmission must account for differing data formats so that all computers on the network understand the data exchanged between them. This capability is often built into the programming library, which first transforms the data into a common format that can be easily decoded on the receiver's side. For this reason, among others, a version of the distributed programming environment must exist for every type of machine architecture participating in the computation.

- Finally, once an application is complete, some way of terminating all the processes must be available.

Many distributed programming environments have received attention in the last five years, including p4 [Butl94], Express [Flow94], Linda [Carr94], and TCGMSG [Harr91]; however, by far the most popular systems are PVM and MPI. PVM was developed at Emory University and Oak Ridge National Laboratory and was first released in 1991. The MPI standard, an international effort, was introduced in 1993.

Both the PVM and MPI message-passing environments, freely available on the world-wide web, provide common interfaces for communication on both multiprocessors and networks of workstations. Both run on many different machines, allowing networked computers of diverse architectures to emulate a distributed-memory multiprocessor. Each of the machines in the

network may be a single-processor or multiprocessor system.

A running process in either PVM or MPI can view a network of computers as a single, virtual machine, ignoring architectural details, or as a set of specialized processors with unique computational abilities. The master process runs on a single computer from which other tasks participating in the computation are initiated. Processes, or tasks, roughly correspond to UNIX processes and operate independently and sequentially, performing both communication and computation. Multiple tasks can run on multiple machines, on a single machine, or a combination of the two.

PVM and MPI are not programming languages; rather, they provide libraries specifying the names, parameters, and results of Fortran and C routines used in message passing. Any program making use of these routines can be compiled with standard compilers by linking in the PVM or MPI library. Note that the developer controls the parallelism in the program by writing master and slave tasks and explicitly specifying the high-level message passing protocol between them. Both PVM and MPI support functional parallelism, in which each task is assigned one function of a larger process; data parallelism, in which identical tasks solve the same problem but for small subsets of the data; or a combination of either approach.

Bibliography

[Acke93] K. Ackeley, "RealityEngine Graphics," *Proceedings of SIGGRAPH 93* in *ACM Computer Graphics*, Aug. 1993, pp. 109-116.

[Akti94] M. Aktihanoglu, B. Ozguc, and C. Aykanat, "MARS: A Tool-based Modeling, Animation, and Parallel Rendering System," *The Visual Computer*, Vol. 11, No. 1, 1994, pp. 1-14.

[Bado94] D. Badouel, K. Bouatouch, and T. Priol, "Distributing Data and Control for Ray Tracing in Parallel," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, 1994, pp. 69-77.

[Butl94] R. Butler and E. Lusk, "Monitors, Messages, and Clusters: the p4 Programming System," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 547-564.

[Carr94] N. Carriero, D. Gelernter, T. Mattson, and A. Sherman, "The Linda Alternative to Message-Passing Systems," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 633-655.

[Clea86] J. G. Cleary, G. Wyvill, and B. M Birtwistley, "Multiprocessor Ray Tracing," *Computer Graphics Forum*, Vol. 5, No. 1, 1986, pp. 3-12.

[Cros95] R. A. Cross, "Interactive Realism for Visualization Using Ray Tracing," *Proceedings of the 1995 IEEE Visualization Conference*, Atlanta, GA, 1995, pp. 19-26.

[Crow88] F. C. Crow, G. Demos, J. Hardy, J. McLaughlin, and K. Sims, "3D image Synthesis on the Connection Machine," *Proceedings of the International Conference on Parallel Processing for Computer Vision and Display* in *Parallel Processing for Computer Vision and Display*, P. M. Dew, T. R. Heywood, and R. A. Earnshaw (Eds.), Addison-Wesley, 1988, pp. 254-269.

[DeMa92] J. M. De Martino and R. Kohling, "Production Rendering on a Local Area Network," *Computers and Graphics*, Vol. 16, No. 3, 1992, pp. 317-324.

[Ells94] D. Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 33-40.

[Evan92] Evans and Sutherland Computer Corporation, *Freedom Series Technical Report*, Salt Lake City, Utah, October, 1992.

[Eyle88] J. Eyles, J. Austin, H. Fuchs, T. Greer, and J. Poulton, "Pixel-Planes 4: A Summary," *Advances in Computer Graphics Hardware II*, Springer-Verlag, Berlin, 1988, pp. 183-207.

[Flow94] J. Flower and A. Kolawa, "Express Is Not Just a Message Passing System-Current and Future Directions in Express," *Parallel Computing*, Vol. 20, No. 4, 1994, pp. 597-614.

[Fuch89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *ACM Computer Graphics*, Vol. 23, No. 3, 1989, pp. 79-88.

[Gaud88] S. Gaudet, R. Hobson, P. Chilka, and T. Calvert, "Multiprocessor Experiments for High-Speed Ray Tracing," *ACM Transactions on Graphics*, Vol. 7, No. 3, 1988, pp. 151-179.

[Geis94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[Gier93] C. Giertsen and J. Petersen, "Parallel Volume Rendering on a Network of Workstations," *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, 1993, pp. 16-23.

[Goel96] V. Goel and A.Mukherjee, "An Optimal Parallel Algorithm for Volume Ray Casting," *The Visual Computer*, Vol. 12, No. 1, 1996, pp. 26-39.

[Gree90] S. A. Green and D. J. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," *The Visual Computer*, Vol. 6, No. 2, 1990, pp. 62-73.

[Gree91] S. A. Green, *Parallel Processing for Computer Graphics*, MIT Press, Cambridge, MA, 1991.

[Grop94] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI – Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA, 1994.

[Harr91] R. J. Harrison, "Portable Tools and Applications for Parallel Computers," *International Journal of Quantum Chemistry*, Vol. 40, 1991, pp. 847-863.

[Henn96] M. Henne, H. Hickel, E. Johnson, and S. Konishi, "The Making of *Toy Story*," *IEEE COMPCON '96 Digest of Papers*, IEEE Computer Society Press, Los Alamitos, CA, 1996, 463-468.

[Hwan93] K. Hwang, *Advanced ComputerArchitecture – Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.

[Isle91] V. Isler and B. Ozguc, "Fast Ray Tracing 3D Models," *Computers and Graphics*, Vol. 15, No. 2, 1991, pp. 205-216.

[Jeva89] D. A. Jevans, "Optimistic Multi-Processor Ray Tracing," *New Advances in Computer Graphics (Proceedings of Computer Graphics International '89*, R. P. Earnshaw and B. M. Wyvill (Eds.), Springer-Verlag, Berlin, 1989, pp. 507-522.

[Keat95] M. J. Keates and R. J. Hubbold, "Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer," *Computer Graphics Forum*, Vol. 14, No. 4, 1995, pp. 189-202.

[Kim96] H-J. Kim and C-M Kyung, "A New Parallel Ray-Tracing System Based on Object Decomposition," *The Visual Computer*, Vol. 12, No. 5, 1996, pp. 244-253.

[Kubo93] Kubota Pacific Computer, *Denali Technical Overview*, version 1.0, Santa Clara, CA, March 1993.

[Lin91] T. T. Y. Lin and M. Slater, "Stochastic Ray Tracing Using SIMD Processor Arrays," *The Visual Computer*, Vol. 7, No. 4, 1991, pp. 187-199.

[Ma93] K-L. Ma and J. S. Painter, "Parallel Volume Visualization on Workstations," *Computers and Graphics*, Vol. 17, No. 1, 1993, pp. 31-37.

[Maur93] H. Maurel, Y. Duthen, and R. Caubet, "A 4D Ray Tracing," *Proceedings of Eurographics '91* in *Computer Graphics Forum*, Vol. 12, No. 3, 1993, pp. 285-294.

[Meno94] J. Menon, R. J. Marisa, and J. Zagajac, "More Powerful Solid Modeling through Ray Representations," *IEEE Computer Graphics and Applications*, Vol. 14, No. 3, 1994, pp. 22-35.

[Moln92] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Proceedings of SIGGRAPH 92* in *ACM Computer Graphics*, Vol. 26, No. 2, pp. 231-240.

[Moln94] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 23-32.

[Nish83] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," *Proceedings of the 10th Symposium on Computer Architecture (SIGARCH)*, 1983, pp. 387-394.

[Pan96] Z. Pan, J. Shi, and M. Zhang, "Distributed Graphics Support for Virtual Environments," *Computers and Graphics*, Vol. 20, No. 2, 1996, pp. 191-197.

[Pito93] P. Pitot, "The Voxar Project," *IEEE Computer Graphics and Applications*, Vol. 13, No. 1, 1993, pp. 27-33.

[Plun85] D. J. Plunkett and M. J. Bailey, "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed," *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, 1985, pp. 52-60.

[Potm89] M. Potmesil and E. M. Hoffert, The Pixel Machine: A Parallel Image Computer," *Proceedings of SIGGRAPH '89* in *ACM Computer Graphics*, Vol. 23, No. 3, 1989, pp. 69-78.

[Prio88] T. Priol and k. Bouatouch, "Experimenting with a Parallel Ray-Tracing Algorithm on a Hypercube Machine," *Proceedings of Eurographics '88*, D. A. Duce and P. Jancene (Eds.), North-Holland, 1988, pp. 243-259.

[Prio89] T. Prioland K. Bouatouch, "Static Load Balancing for Parallel Ray Tracing on a MMD Hypercube," *The Visual Computer*, Vol. 5, Nos. 1 and 2, 1989, pp. 109-119.

[Reis97] A. Reisman, C. Gotsman, and A. Schuster, "Parallel Progressive Rendering of Animation Sequences at Interactive Rates on Distributed-Memory Machines," *Proceedings of the 1997 IEEE Parallel Rendering Symposium*, Phoenix, AZ, 1997, pp. 39-47.

[Ris94] P. Ris and D. Arques, "Parallel Ray Tracing Based upon a Multilevel Topological Knowledge Acquisition of the Scene," *Proceedings of Eurographics '94* in *Computer Graphics Forum*, Vol. 13, No. 3, 1994, pp. 221-232.

[Schr92] P. Schroder and S. M Drucker, "A Data Parallel Algorithm for Raytracing of Heterogeneous Databases," *Proceedings of Computer Graphics Interface '92*, Vancouver, British Columbia, 1992, pp. 167-175.

[Shen95] L-S. Shen, E. F. Deprettere, and P. Dewilde, "A Parallel Image-Rendering Algorithm and Architecture Based on Ray Tracing and Radiosity Shading," *Computers and Graphics*, Vol. 19, No. 2, 1995, pp. 281-296.

[Silv94] C. T. Silva and A. E. Kaufman, "Ray Parallel Performance Measures for Volume Ray Casting," *Proceedings of the 1994 IEEE Visualization Conference*, Washington, D. C., 1994, pp. 196-203.

[Sous90] A. A. Sousa, A. M. C. Costa, and F. N. Ferreira, "Interactive Ray-Tracing for Image Production with Increasing Realism," *Proceedings of Eurographics '90*, North-Holland, 1990, pp. 449-457.

[Stob88] A. Stober, A. Schmitt, B. Neidecker, H. Muller, T. Maus, and W. Leister, "*'Occursus Cum Novo'* – Tools for Efficient Photo-Realistic Computer Animation," *Proceedings of Eurographics '88*, D. A. Duce and P. Jancene (Eds.), North-Holland, Amsterdam, 1988, pp. 31-41.

[Sung96] K. Sung, J. L. J. Shiuan, and A. L. Ananda, "Ray Tracing in a Distributed Environment," *Computers and Graphics*, Vol. 20, No. 1, 1996, pp. 41-49.

[Whit92] S.Whitman, *Multiprocessor Methods for Computer Graphics Rendering*, Jones and Bartlett Publishers, Boston, 1992.

[Whit94] S.Whitman, "A Task Adaptive Parallel Graphics Renderer," *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 41-48.

[Whit80] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Vol. 23, No. 6, 1980, pp. 343-349.

[Worl93] L. World, "Low-End System Animates the Depths in *SeaQuest*," *IEEE Computer Graphics and Applications*, Vol. 13, No. 6, 1993, p. 93.

Global Illumination in Complex Scenes using Photon Maps

Henrik Wann Jensen
Department of Graphical Communication
Technical University of Denmark

January 5, 1995

Abstract

A general global illumination method is presented which handles complex geometries using the concept of photon maps. The global illumination method is a two pass method in which the first pass consists of emitting photons from the light sources in the model and storing these in the photon map. The photon map is independent of scene geometry, it does not require tessellation of the objects and it can be used with procedural objects and any other objects that can be ray traced. Two layers of photons are used: One approximate global layer used for the simulation of soft indirect illumination on diffuse surfaces and one accurate layer used for the simulation of caustics. In the second pass the scene is rendered using optimized Monte Carlo ray tracing. The recursive nature of Monte Carlo ray tracing is avoided by using the results from the photon map. The method simulates Anisotropic reflection and caustics on complex surfaces and the results achieved has been verified by comparison with a brute force path tracing algorithm.

The method has been parallelized on a network of 31 Silicon Graphics Workstations resulting in a speedup from 20 to 28.

General Terms: Algorithms
Key Words: Global Illumination, Rendering, Ray Tracing, Radiosity, Photon Maps, Monte Carlo, Parallelization.

1 Introduction

The simulation of global illumination has been investigated thoroughly in the last 10 years. The introduction of ray tracing [24] and in particular the introduction of radiosity [7, 5] has been the foundation on which current global illumination methods are based. Neither of the two methods satisfy the needs in global illumination. Ray tracing only simulates ideal specular reflection while radiosity only handles ideal diffuse reflection.

In 1986 Kajiya [11] introduced the path tracing method which is an extension of the ray tracing method with Monte Carlo techniques. Path tracing is still today the most general global illumination solution. It can simulate global illumination in arbitrarilly complex models both with respect to geometry and surface properties. Unfortunately the computational cost of path tracing is enormous and effects like caustics are only simulated properly in theory. Ward et al. [19, 22] improved path tracing by caching information at ideal diffuse surfaces and newer bidirectional path tracing algorithms [12] has improved the simulation of indirect illumination. These extensions does, however, not adequately solve the caustics problem.

The radiosity algorithm has also been extended towards the simulation of specular reflection. Immel et al. [9] created a form factor with directional information. This approach is however almost as complex as path tracing and it uses enormous amounts of memory even for simple scenes. Sillion et al. [17] improved this method significantly by using spherical harmonics to compress the directional information stored in the model and by using ray tracing to simulate ideal specular reflection.

Currently the most successful approaches are two pass methods in which the first step is emission of light from the light sources (for example progressive radiosity and light ray tracing [1]). The result of this step is irradiance or radiosity values stored in either a polygon model or an illumination map. The second step is visualization of the model. This step is generally carried out using distribution ray tracing and the irradiance values computed in the first pass are reused during rendering. The idea of a two pass approach was introduced in 1987 by Wallace et al. [18]. They used a radiosity algorithm extended to simulate a limited number of specular reflections. The solution was rendered directly using an optimized distribution ray tracing algorithm that only calculated specular reflections. Shirley [15] improved the simulation of caustics and shadows using ray tracing. Sillion et al. [17] improved the first pass via a progressive radiosity algorithm extended with the simulation of directional diffuse reflection. Chen et al. [4] presented a very comprehensive global illumination method in which visible artifacts from the radiosity step is eliminated by using path tracing to simulate all reflections seen directly by the eye — including the first diffuse reflection.

One of the reasons behind the success of the two pass approaches are the storage of irradiance/radiosity values within the model. This storage contains precalculated and view-independent information on how the model is illuminated by the light sources in the model. This includes indirect illumination such as caustics and color bleeding. In general these terms are computed most efficiently by starting at the light sources, like progressive radiosity and light ray tracing. Calculating color bleeding using path tracing from the eye often requires several thousand rays pr. pixel and caustics from small light sources are often almost impossible to calculate by stochastic techniques from the eye. In the visualization pass of the two pass methods the most successful technique is a simplified path tracing method or distributed ray tracing since it quite effectively simulates specular effects and sharp shadows. The calculation of indirect illumination can be done simply by looking up information in the storage and this leads to a quite efficient approach.

However the storage in the two pass approaches introduces a problem when it comes to the simulation of global illumination in geometrically complex models. For example models containing procedural objects or models containing several thousand instantiated version of one complex object (e.g. a lawn created by copying a piece of grass). The reason is that the storage techniques used requires a representation of every single object in the model. The radiosity algorithm in its current form stores radiosity values with each polygon in the model and illumination map techniques requires the ability to connect an illumination map to every simple object in the model. This effectively limits the number of objects that can be used. Currently only path tracing and it's derivatives can simulate global illumination in arbitrarilly

Method	Author & Paper	CPU-time	Memory Usage	Arbitrary Geometry	Arbitrary BRDF	Caustics
Path tracing	Kajiya [11]	Very high	Low	Yes	Yes	Limited
Bidir. path tracing	Lafortune et al. [12]	High	Low	Yes	Yes	(Limited)
Radiance	Ward [23]	Medium	Medium	Yes	Yes	From polygons
Two pass	Shirley [15]	Medium/Low	Medium	No	No	On simple objects
Two pass	Sillion et al. [17]	Medium	High	No	Yes	(No)
Two pass	Chen et al. [4]	Medium	Medium	No	(Yes)	On simple objects

Table 1: Global illumination methods.

complex environments — unfortunately at great computational cost.

We have tried to classify some of the existing global illumination methods. This is difficult since each of the methods has merits that cannot be put into a table. In table 1 we have shown a few characteristics of 3 Monte Carlo based methods and 3 two pass methods. In general one can conclude that very complex geometries cause the Monte Carlo methods to use a lot of CPU-time while the two pass methods use a lot of memory.

We would like to create a two pass method in which the storage of irradiance is independent of the geometry in the model in order to prevent having to store irradiance values with every surface in the scene. In [10] we presented a global illumination method in which we stored irradiance values in illumination maps on most surfaces in the scene. On complex objects like fractals we stored irradiance in photon maps. This new concept allowed us to represent irradiance upon arbitrary geometries. The basic component in the photon map is the photon which represents a packet of energy striking a surface. This packet of energy and the intersection point is stored in the photon map. The irradiance at specific positions in the scene is estimated from the nearest photons in the photon map. The photon map was only used to simulate irradiance on complex objects in order to reduce the number of photons. In complex scenes containing many complex objects the number of photons required would be very large and the memory needs would exceed that of radiosity. The main reason for this was that only one photon map was used to simulate both caustics and irradiance in general. This meant that the accuracy in the photon map had to be very high requiring a large number of photons. This paper presents a enhanced two pass methods in which two photon maps are used. One accurate photon map is used to simulate caustics on complex surfaces and one approximate photon map is used to simulate indirect illumination on diffuse surfaces. This significantly reduces the number of photons needed since we only have to be accurate in the photon map simulating caustics — this also means that we do not have to use illumination maps.

In [10] only ideal diffuse and ideal specular reflection was simulated. We extend this with the simulation of anisotropic reflection still without restrictions on the geometric complexity.

We also test a simple parallelization scheme with the two pass method. Currently only little research has been put into parallelizing two pass methods. This can to some extent be explained by the complexity of existing methods combined with the fact that the methods needs to exchange information globally in the model (ie. radiance values computed in one part of the scene is used to compute the radiance values in other parts of the scene). A notable exception is the path tracing techniques in which each ray can be evaluated independently of the other rays — the number of rays is unfortunately so large that a huge number of processors is needed to make the method tractable. Ward [23] uses shared memory to exchange global radiance information among a number of individual workstations. This shared memory will unfortunately become a bottleneck as more workstations are added and it limits the degree of parallelization achievable. Using a copy of the photon map at each processor in the network allows us to parallelize the method without using shared memory and with only very little communication. The photon map prevents massive re-computation of the same irradiance values on each processor so we can expect a reasonable speedup.

2 Simulating Global Illumination using Photon Maps

This section contains a detailed description of the two pass method. The method was constructed using the following requirements:

- It must handle arbitrary geometries
- It should be capable of simulating surfaces with an arbitrary BRDF
- It must simulate all light paths
- It should not be too demanding with respect to memory and CPU-resources.
- It should be possible to perform the calculations on several workstations.

Based upon these requirements we have to use a Monte Carlo based approach since currently this is the only method that can handle arbitrary geometries.

We use a two pass method in which the first pass consists of emitting photons from the light sources in the model towards the objects. The technique used is photon tracing in which photons are followed along a path and stored at each object intersection. Two photon tracing steps are used: One accurate in which photons are emitted towards the specular objects in the scene and one less accurate where photons are emitted towards all objects in the scene. The second pass is optimized Monte Carlo ray tracing. This pass renders all reflections seen directly by the eye with the exception of caustics which are taken directly from the photon map. The indirect illumination on diffuse surfaces is also taken from the photon map and this eliminates the explosion of rays that occurs in a naive Monte Carlo based renderer.

In the following sections we use the terms specular and diffuse reflection. These are not restricted to ideal specular or ideal diffuse reflection but they characterize the type of reflection. Specular reflections are very directional while diffuse reflection does not depend much on directions. Furthermore we do not discuss transmission which is treated analogously to reflection.

2.1 Pass 1: Photon Tracing

In the photon tracing pass particles, photons, are emitted from the light sources in the scene. Each photon carries a

fraction of the light source energy and it follows a straight path until it hits an object. When a photon hits an object it deposits energy into the photon map and using Russian roulette [2] it is decided whether the photon is reflected or absorbed by the surface of the object. If the photon is reflected the BRDF of the surface is used to determine the new direction of the reflected photon. If the BRDF of the surface is a combination of several BRDFs (e.g. specular and diffuse) then Russian roulette is used to select one of these and this BRDF is used in the calculation of the reflected direction — the energy of the photon is scaled accordingly. This means that we only follow one single path and completely avoids an explosion in the number of photons.

Storing each photon is costly since a large number of photons must be used in order to simulate effects such as caustics properly. Caustics are visualized directly and they require a large number of photons in order to avoid noise in the final solution. This accuracy is, however, not necessary in the rest of the photon map since this is only used to simulate soft indirect illumination (more on this in section 2.2). This gives rise to an important optimization that can reduce the number of photons needed dramatically. We simply have to use two photon maps, one carrying photons that are part of a caustic and one photon map in which the photons of a complete, but less accurate solution is stored.

In order to avoid shooting photons in all directions around each light source we use the projection map [10] to select the important directions. The projection map is a hemisphere on which the positions of the diffuse and specular objects in the scene relative to the position on the light source, from which photons are being shoot, are marked. It is similar to the light buffer [8] used to optimize shadow sampling. The projection map requires projection of the complete model onto a hemisphere which is to costly if the scene is very complex. In this case a stochastic technique like the one described by Shirley [15] could be used.

Photons are emitted from different positions on each light source. At small light sources only one position is used. This allows us to optimize the emission of photons via stratification of the (hemi)sphere around the light source. If large area light sources are used then the same strategy can be applied using more positions on the light source.

2.1.1 Creating the caustic photon map

The photons are only emitted towards the specular objects in the scene and the number of photons emitted in these directions are determined by the properties (e.g. specularity) of the surface. This means that more photons are emitted in directions with highly specular objects.

When a photon hits an object it is decided whether this photon should deposit energy into the photon map (ie. the photon is part of a caustic). This is done if the photon has been reflected specularly at least once and the BRDF of the surface has a diffuse component. Russian roulette is then used to decide whether the photon should be absorbed or reflected. If the BRDF of surface has both a diffuse component and a specular component then Russian roulette is used to select one of them. The path of the photon is only followed if it is reflected specularly. If the photon is absorbed or reflected diffusely then it is terminated.

2.1.2 Creating the global photon map

In this step the photons are emitted towards all objects in the scene and the density of photons emitted in the different directions only depend on the emission characteristics of the light source.

When a photon hits an object the energy of the photon is deposited into the photon map if the BRDF of the surface has an diffuse component. The photon is then absorbed or reflected depending of the outcome of the Russian roulette algorithm. If reflected the new direction of the photon is determined using the BRDF of the surface.

This results in a coarse sweep of photons across the surfaces in the scene. There is no guaranty that all objects will be hit (if they are small) and the sweep as such represents only an approximate simulation of the global illumination in the scene.

2.2 Pass 2: Rendering

The rendering pass requires high precision since the result is visualized directly to the eye. The scene is rendered using optimized Monte Carlo ray tracing in which the pixel radiance is computed by averaging a number of sample estimates. Each sample consists of tracing a ray from the eye through the pixel based on some filtering function. The radiance value returned by each ray equals the radiance, $L_o(\mathbf{x}, \theta_o, \phi_o)$, leaving the first surface intersected by the ray. $\mathbf{x}$ is the intersection point and (θ_o, ϕ_o) is the direction of the ray. $L_o(\mathbf{x}, \theta_o, \phi_o)$ can be found by summing the emitted radiance, $L_e(\mathbf{x}, \theta_o, \phi_o)$, the reflected radiance, $L_r(\mathbf{x}, \theta_o, \phi_o)$ and the transmitted radiance, $L_t(\mathbf{x}, \theta_o, \phi_o)$.

$$L_o(\mathbf{x}, \theta_o, \phi_o) = L_e(\mathbf{x}, \theta_o, \phi_o) + L_r(\mathbf{x}, \theta_o, \phi_o) + L_t(\mathbf{x}, \theta_o, \phi_o) \tag{1}$$

L_e is taken directly from the surface definition and needs no further calculation. L_r and L_t depend on the radiance values in the rest of the scene and requires further computation. They are treated analogously and in the following discussion only the computation of L_r is described.

The reflected radiance is found by integrating the incoming radiance, $L_i(\mathbf{x}, \theta_i, \phi_i)$, multiplied by the bidirectional reflectance distribution function, $f_r(\mathbf{x}, \theta_i, \phi_i; \theta_o, \phi_o)$, over the hemisphere of incoming directions:

$$L_r(\mathbf{x}, \theta_o, \phi_o) = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} f_r(\mathbf{x}, \theta_i, \phi_i; \theta_o, \phi_o) L_i(\mathbf{x}, \theta_i, \phi_i) cos\theta_i \, d\omega_i \tag{2}$$

This integral can be solved directly using Monte Carlo techniques (e.g. path tracing), but as stated earlier this leads to a very expensive solution. A more efficient approach can be obtained by using our knowledge of the BRDF, and the incoming radiance. Efficient techniques exist for calculating specular reflections, contributions from the light sources and caustics and it would be beneficial to split equation 2 in order to allow these contributions to be calculated separately. Omitting the position and direction parameters for clarity results in the following formulation for the calculation of L_r:

$$\begin{aligned} L_r \quad = \quad & \int\int_\Omega f_r L_{i,l} \cos\theta_i \, d\omega_i + \\ & \int\int_\Omega f_{r,s} (L_{i,c} + L_{i,d}) \cos\theta_i \, d\omega_i + \\ & \int\int_\Omega f_{r,d} L_{i,c} \cos\theta_i \, d\omega_i + \\ & \int\int_\Omega f_{r,d} L_{i,d} \cos\theta_i \, d\omega_i \end{aligned} \tag{3}$$

where

$$f_r = f_{r,s} + f_{r,d} \quad \text{and} \quad L_i = L_{i,l} + L_{i,c} + L_{i,d}$$

In this equation the incoming radiance has been split into a contribution from the light source, $L_{i,l}$, a contribution from the light source via specular reflection (caustics), $L_{i,c}$ and indirect soft illumination, $L_{i,d}$ which has been reflected diffusely at least once. The BRDF has been separated into a

diffuse part, $f_{r,d}$, and a specular part, $f_{r,s}$. In the following sections we discuss the evaluation of each of the parts in equation 3.

2.2.1 $\int\int_\Omega f_r L_{i,l} \cos\theta_i \, d\omega_i$

This term represents the direct light contribution from the light sources in the scene. It is in principle easily determined by sending shadow rays towards the light sources in order to check for visibility and adding the contribution from the visible light sources. However this strategy can be very costly requiring a large number of rays in scenes with many light sources. [16] has optimized this by using a fixed number of shadow rays. [20] uses an adaptive light sampling approach. Both methods can be used here. Currently we use a mixture of the adaptive light sampling approach and simple stratified sampling of the light sources.

2.2.2 $\int\int_\Omega f_{r,s}(L_{i,c} + L_{i,d}) \cos\theta_i \, d\omega_i$

Specular reflection is efficiently computed using standard Monte Carlo ray tracing in which a limited number of rays are used to sample the contribution from the specular direction. The number of rays used depends on the specularity of the surface. For ideal specular surfaces only one ray has to be used. For more complex anisotropic surfaces the number of rays depends on the roughness and specularity of the surface (see section 2.2.5).

2.2.3 $\int\int_\Omega f_{r,d} L_{i,c} \cos\theta_i \, d\omega_i$

The computation of caustics is done simply by estimating the irradiance value using the caustic photon map. See section 2.3 for an explanation on how this is done.

2.2.4 $\int\int_\Omega f_{r,d} L_{i,d} \cos\theta_i \, d\omega_i$

This term integrates incoming light which has been reflected diffusely at least once since it left the light source. This light is then reflected diffusely by the surface (using $f_{r,d}$) and consequently the resulting illumination is very soft. If $f_{r,d}$ is ideal diffuse then the irradiance gradient method proposed by Ward et al. [22] is used. This method is well suited since it is only used to model very soft indirect illumination. If $f_{r,d}$ is directional diffuse or more complex then we use a Monte Carlo based approach in the evaluation.

The evaluation of the indirect illumination has been optimized significantly using the global photon map. When any of the sampling rays used in this pass hits an ideal diffuse surface the complete irradiance value is taken from the global photon map. This includes the direct light from the light sources and therefore we do not have to use shadow rays.

2.2.5 Simulating Anisotropic Reflection

In 1992 Ward [21] presented a BRDF formula for simulating anisotropic reflection. The anisotropy was simulated combining ideal diffuse reflection with rough specular anisotropic reflection and it directly provides a BRDF which has been split into a diffuse and a specular part and this BRDF can be used directly in our model (see [21] for a formulation of the formulas).

To achieve a good visualization of the anisotropic surfaces Ward proposes a hybrid sampling in which the light sources are sampled separately from the rough specular component. Using equation 3 we directly achieve this sampling strategy. The rough specular reflection is sampled using Monte Carlo ray tracing based on the formulas also given in Wards paper. The number of samples used depend on the specularity and the roughness of the surface.

In the photon tracing pass the density of the photons emitted towards a surface with anisotropic reflection is likewise determined from the roughness and the specularity of the surface.

2.3 The Photon Map

The photon map represents an alternative way of storing the irradiance within a scene. In the photon map the basic element is the photon which corresponds to a packet of energy emitted from the light source and perhaps reflected by objects in the scene.

With every photon we store the energy, the intersection point (normal and position). We also need two pointers for the data structure which is a kd-tree [3]. The kd-tree has been chosen since we need to be able to locate photons within a given volume in order to estimate the irradiance. An alternative data structure could be a Voronoi diagram.

How can we estimate the irradiance given just a number of points in a volume? Actually we are interested in computing the radiance leaving a small area dA:

$$L_r = \int\int_\Omega f_r L_i \cos\theta_i \, d\omega_i = \int\int_\Omega f_r \frac{d^2\Phi(\theta_i, \phi_i)}{dA}$$

This equation states that in order to compute the reflected radiance from a surface we must integrate the incident radiant flux, $d\Phi(\theta_i, \phi_i)$, on the surface. In order to accomplish this a number of assumptions are necessary. If the reflection of light from dA is ideal diffuse then f_r becomes constant and we get:

$$L_r = \frac{\rho_d}{\pi} \int\int_\Omega \frac{d^2\Phi(\theta_i, \phi_i)}{dA}$$

where ρ_d is the diffuse reflectance.

If we have a small area ΔA and we know that N photons each represent a small packet of incident radiant flux Φ_e onto ΔA then we can approximate the integral with the following sum.

$$L_r \approx \frac{\rho_d}{\pi} \left(\frac{\sum_{n=1}^{N} \Phi_{e,n}}{\Delta A} \right)$$

Thus the irradiance is estimated as

$$E_i = \frac{\sum_{n=1}^{N} \Phi_{e,n}}{\Delta A}$$

In order to determine the irradiance at a position $\mathbf{x}$ we need to find N photons and determine the area ΔA. The N photons are located as the N photons with the smallest distance to $\mathbf{x}$ with the requirement that the dot product between normal stored with the photon stored in the photon map and the normal at $\mathbf{x}$ must be positive — to ensure that we do not get light leaking from photons on the backside of flat objects.

The area ΔA is estimated using the maximum distance R between $\mathbf{x}$ and the photons. ΔA is computed as the area of a circle with radius R:

$$\Delta A = \pi R^2$$

This approximation is reasonable when the surface on which the photons are located is flat but not as good when the surface is very rough or if $\mathbf{x}$ is close to an edge. If the number of photons emitted from the light source is to small this is seen as blurred edges. The solution converges towards the exact one as the number of photons emitted from the light source goes to infinity.

The argument of the computation of E_i could be turned around if we start by looking at a small circle area. This is actually a sphere volume since the photons are located in

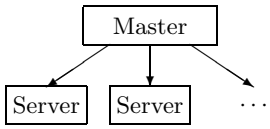


Figure 1: Master-server process-configuration.

3d-space. However if the area is small and we are at the surface of an object we can assume that we have a circle. In order to compute the irradiance onto that circle we simply count the number of photons reflected (= the outgoing flux) by the circle and divide this by the area of the circle.

Our final problem is locating the N photons around $\mathbf{x}$. The kd-tree is well suited when it comes to range searching and given a position $\mathbf{x}$ we can relatively easy locate the nearest photons stored in the kd-tree. However since the kd-tree is constructed on the fly by photons reflected in the scene we risk getting a very skew kd-tree resulting in longer search time. We have found that pruning the search by initially limiting the distance from $\mathbf{x}$ improves the speed of the search dramatically. This distance could be determined adaptively based on the distance of previous searches. We use a fixed distance which is a parameter to the model for example we could say that photons that are further away than 0.1 m. should be ignored. For normal-sized models this distance is reasonable because if the distance is larger the density of photons onto the surface is too low to give any significant contribution. We use two different distance limits for the two different photon maps.

3 Parallelization of the Algorithm

In order to make the method more tractable in interactive applications we decided to run it in parallel on a network of 31 Silicon Graphics Workstations each equipped with 24 MB RAM and a 33MHz MIPS R3000 processor. We decided to focus on the rendering stage which is the most demanding part of the computation.

Our hardware is not aimed at parallelization and it has only limited capacity when it comes to communication so we want to exchange as little information as possible. Furthermore we would like to see the result being presented progressively as computations are finished. We therefore decided to use a simple master-server configuration as shown in figure 1.

The method has been parallelized as follows. The master sends jobs to the servers and receives and displays results. Each server has a full copy of the scene and the photon map. Initially each server reads the scene description and performs pass 1, the photon tracing step. This results in a local copy of the photon maps. The server then reports back to the master which in return sends a job to the server. A job is an area of the screen and the server must compute the pixel radiance values within this area. When finished the results are send to the master which updates the display and sends a new job to the server. This continues until the results from all jobs have been received by the master and the image is complete.

Using this scheme we risk computing radiance values in the same part of the scene more than once. However due to the construction of the method we do not get an explosion in the number of radiance values computed since the photon map replaces most computations. The irradiance values computed using the irradiance gradient method are saved from one job to the next on each server allowing some re-usage of the irradiance values.

The prime advantage of preventing communication between the servers or perhaps using shared memory to store the irradiance values is that we do not limit the number of workstations that can be used with the method. The master of course represents a potential bottleneck since it has to receive all the results. If a lot of workstations were available it would probably be beneficial to introduce sub-masters that could receive large jobs from the master and spread these onto a limited range of servers attached to each sub-master.

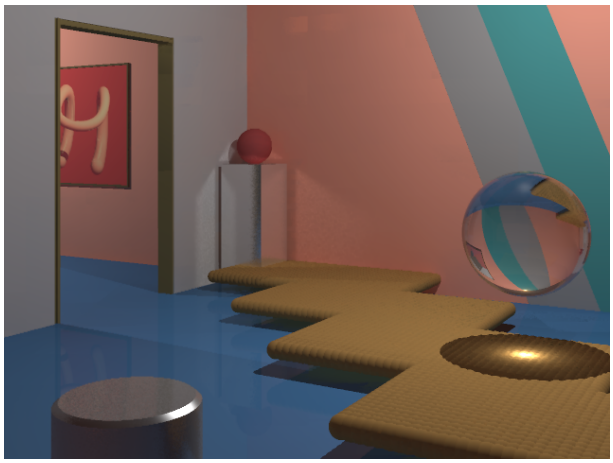


Figure 2: The museum

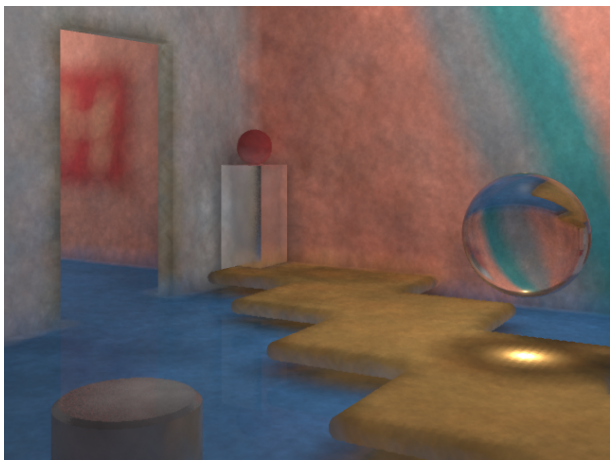


Figure 3: The rough photon map in the museum visualized directly.

4 Results

The global illumination method described has been developed on a 486DX2-66 PC with 32MB RAM running Linux (a Unix-clone) and the following results have been produced using this machine. The images have been rendered in 640x480 with 4 samples pr. pixel.

The first test scene is shown in figure 2. It shows the interior of a museum containing a number of exhibits. The cube in the corner has anisotropic reflection. It has been polished in the direction from the floor towards the ceiling. The top of the cube has isotropic (rough specular) reflection. The cylinder in front also demonstrates anisotropic reflection. The sides of the cylinder has been polished horizontally round the cylinder. The top of the cylinder is isotropic. The glass sphere is transmitting and reflecting light in accordance with

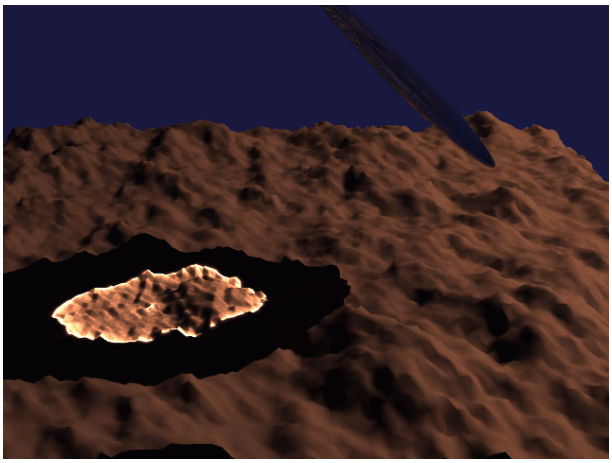


Figure 4: Fractal landscape with caustic

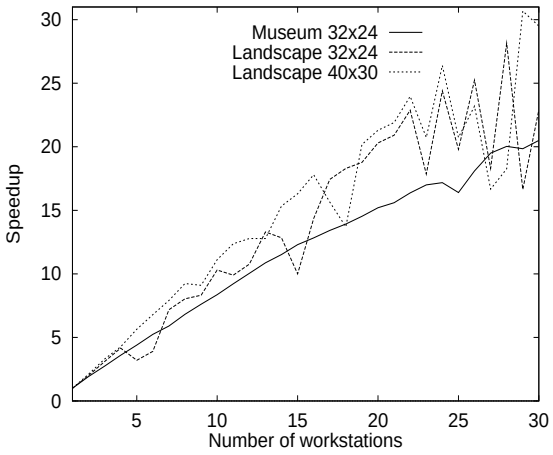


Figure 5: The result of parallelization of the method.

the Fresnel formula for dielectric materials. The collection of spheres on the floor has been created using one square shaped element consisting of 400 intersecting spheres. This element has been reused 9 times in order to create the sphere art on the floor. Two color stripes has been painted on the back wall. The floor is slightly specular. The scene contains two small light sources one in the room seen and one in the neighboring room.

The museum was rendered in 2:25 hours and pass 1 took 3:27 minutes. We used 143.949 photons in the caustic photon map and 83.802 photons in the rough photon map — this correspond to a memory usage of approx. 6.4 MB with 28 bytes pr. photon. The final image is shown in figure 2. This image has several caustic effects: On the wall due to anisotropic reflection, on the bottom of the red sphere due to isotropic (rough specular) reflection, and on the sphere art on the floor due to transmission through the glass sphere. The specularity of the floor does not give any visible caustics even though it has been illuminated as a part of the creation of the caustics photon map. The image also demonstrates color bleeding, in particular on the partitioning wall at the left which is painted white but due to color bleeding looks both a little blue and red. Rough specular reflection from the anisotropic and isotropic materials are also demonstrated. The top of the cylinder gives a rough specular reflection of the image in the neighboring room.

We also visualized the global photon map directly and the result is shown in figure 3. In this image no direct light or indirect light has been computed — only specular reflection. The irradiance values have been taken directly from the rough photon map. The overall illumination in the image is clearly the same as in the final image. The looks are however more noisy and some errors occur at the edges but this noise is not visible in the final image.

The museum image was also calculated using a recursive Monte Carlo ray tracing approach in which only the irradiance gradient method was used. This test was performed using few samples pr. pixel and the rendering time was 6:59 hours for the irradiance gradient method while our approach took only 1:29 hours.

Our second test (fig. 4 scene demonstrates the simulation of caustics on a procedural object. The procedural object is a fractal landscape created via discrete sampling of Perlins Noise Function. The landscape is illuminated from a spherical light source both directly and through a lens shaped ellipsoid giving rise to a caustic on the landscape. Notice how the illumination of the landscape within the caustics looks similar to the illumination of the landscape caused by direct illumination.

The image was rendered in 8.2 minutes and pass 1 took 7.0 minutes. The caustic photon map contains 180.756 photons and the rough photon map contains only 12.550 photons — this is adequate since the model contains only very little indirect illumination. This image particularly tests the rendering of caustics and the rendering time is short since the scene contains very little indirect illumination.

4.1 Parallel Implementation

The parallel implementation was realized using one workstation as the master and 30 workstations as servers. Each server was implemented as a process sleeping on each workstation only waiting for jobs from the master. The master has been refined so that it automatically discovers if one of the servers disappear (e.g. the workstation is shut down) and it sends the jobs another one.

We have measured the rendering time and computed the speedup as:

$$S(n) = \frac{T(1)}{T(n)}$$

where $T(n)$ is the rendering time using n servers (workstations). We obtained the speedups shown in the graph in fig. 5. The speedups have been measured on the museum and the landscape scenes. With the landscape scene we tested the effect on speedup using different sizes of jobs (32x24 pixels and 40x30 pixels). The graph demonstrates a nice speedup — using 30 machines we reduce the rendering time of the museum by a factor of approx. 21 and of the landscape by a factor of approx. 25. The fact that the speedup of the museum is lower than that of the landscape is clearly due to the fact that the museum contains more indirect illumination combined with specular surfaces. This means that more irradiance values has to be computed several times. The speedup achieved with the landscape did not give any significant difference with the two different job sizes. The graph also becomes very noisy as the number of workstations grows larger than 20. This is due to the fact that the rendering time using 20 workstations approaches 30 sec. and the communication speed begins to affect the speed of the computations.

In some cases we obtained speedups larger than n! Using 10 workstations the rendering time was reduced with a factor 10.3 and this is not just a coincidence. We observed this several times and we believe that it is due to the fact that the data structure build while rendering (e.g. the irradiance

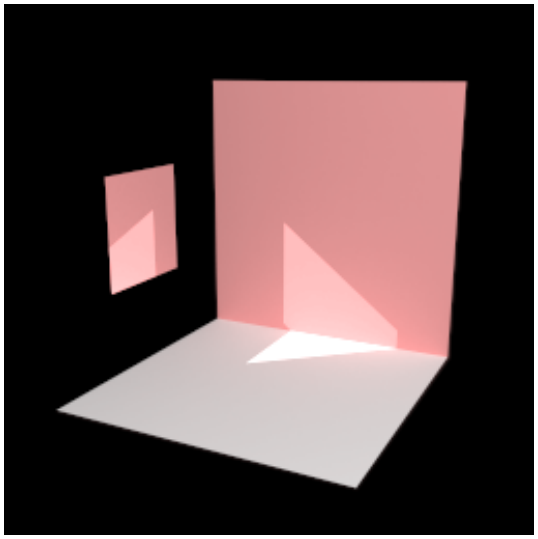


Figure 6: Scene to test method against path tracing

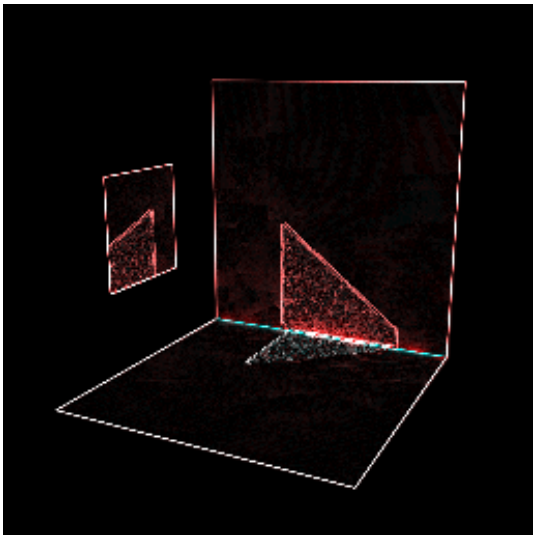


Figure 7: Error image (the error has been magnified 8 times). Notice that there is only very little noise within the caustic. The most erroneous part is the edges where the path tracing algorithm has a higher accuracy.

values computed using the irradiance gradient method) becomes smaller on each server. This reduces the time it takes to locate and reuse irradiance values and it makes the rendering process faster. Using 1 machine the rendering times where 12.2 minutes for the landscape and 174.6 minutes for the museum. Using 30 machines the rendering times reduced to 30 seconds for the landscape and 8.5 minutes for the museum.

4.2 Comparison with the Path Tracing Algorithm

A simple test scene has been designed to allow a comparison between the global illumination method presented and a brute force path tracing solution. This comparison is justified by the fact that it is quite difficult to predict the effect of using the photon map in the final solution.

The test scene is shown in figure 6. It demonstrates both caustics and color bleeding in it and it can be used to test both the caustic photon map and the global photon map. We created a reference image using a simple path tracing algorithm with 5000 rays pr. pixel and 256x256 pixels — the caustic was computed using exact calculations. It took approx. 12 hours to compute this image. Our method used 12 seconds in pass 1 and the rendering in pass 2 took 79 seconds. The rendering was done using only 1 ray pr. pixel. We used 7933 photons in the rough photon map and 13159 photons in the caustic.

Comparing the two images can be done in several ways. Just looking at the two images reveals no difference except for the aliasing problem near the edges in our image. We decided to examine the images by subtracting them thereby obtaining a difference image in which black indicate no difference. In order to reveal small errors we scaled the intensity in the difference image by a factor 8 and we got the results shown in figure 7. This image clearly reveals that the edges has not been sampled properly. It also reveals that even though the error has been scaled by a factor 8 the remaining parts of the image are nearly black (we use floating point to represent the images so this is not just a result of rounding down). There is a little noise in the caustic (that is very difficult to see in our image) and there is also a little noise close to the edge where the red and the white square meet. In corners like this inaccuracies in the photon map can easier affect the irradiance computations.

5 Discussion

Compared to the way the photon map was used in [10] our handling is superior. We use far fewer photons. There is better control of the number of photons due to the separation of the photon map into two maps. This separation has also resulted in better and faster approximation of the indirect illumination. The old method could give some artifacts near edges. This is particularly due to the fact that only one photon map was used. These artifacts have been significantly reduced. The distance criteria also improved the speed of the search during rendering. Using the old technique the landscape in figure 4 took 3 hours and 22 minutes to render. The improved method in this paper uses 8 minutes.

The irradiance estimate in the photon map is based on a few assumptions that can give inaccurate estimates. The size of the area which is hit by photons is estimated based on the assumption that the surface is flat. Since the photons are located in a volume this estimate is poor near edges and on very rough surfaces. These errors are rarely visible in the final images and they can be eliminated by increasing the number of photons in the photon map. Since we use a relatively high number of photons in the caustic photon map we have not observed any artifacts even though it is visualized directly.

Comparison with result achieved using the path tracing algorithm demonstrated only small (non visible) errors in our method. The errors were a little noise in the caustic and a little noise near the edges.

Even though the parallelization of the method is quite simple it gives large speedups. This is due to the fact that each server has a complete photon map. If we had not used the global photon map but only the irradiance gradient method the speedups would clearly become worse since the same irradiance values would be computed on several servers. However a small problem that can arise when using distributed irradiance values. The extrapolation and interpolation in the irradiance gradient method gives only approximate results and different irradiance values and gradients gives different approximations. This can show up as edges in the image. They appear between pixels evaluated on different servers. This problem can be solved by increasing the accuracy of the irradiance gradient method.

It would also be reduced if the master was more intelligent in distributing jobs. Currently news jobs are given to any available server. The master could send jobs belonging to the same part of the screen to the same server. This would allow for better re-usage of the computed irradiance values and also reduce the appearance of edges in the image.

The method does have one problem in common with the rest of the global illumination methods. Scenes which contains many light sources are difficult to handle. The determination of the direct light from these light sources takes a lot of time. Currently we have implemented adaptive stratified sampling of the light sources. Another problem is the emission of photons from many light sources. The photons emitted from each light source has to be lowered if many light sources exists in order to limit the number of photons used. This will probably not cause any problems since one light source competing with many light sources usually only gives important and varying illumination on neighboring objects. In general many light sources would, however, require a more intelligent emission of photons.

Using the global photon map we do not need to use shadow rays in the computation of indirect illumination. This is the case with the irradiance gradient method and preliminary investigations show that our method becomes superior to the pure irradiance gradient method as the number of light sources grow.

If the scene is not very complex but contains many light sources. Then the global photon map could be substituted by a fast radiosity solution on a tessellated version of the scene.

Another area of interest is the irradiance estimation performed in the photon map. Even though it is approximate we believe that it can be extended to handle caustics on surface with directional diffuse properties by storing also the incoming direction of the photon and performing a full numerical integration of the incident radiant flux.

6 Conclusion and future directions

We have presented a two pass global illumination method that can be used in complex scenes. The method does not require tessellation of the model and it can simulate caustics on complex procedural models. The method handles anisotropic, isotropic as well as diffuse and specular reflections.

We use two photon maps to store irradiance values. One for caustics and one approximate used in the computation of indirect illumination on diffuse surfaces. The global photon map can be created using only few photons. This results in irradiance estimates that may very well be noisy. However, this noise is not visible in the final image since the irradiance values are reflected diffusely before reaching the eye.

We have demonstrated that it is possible to achieve good speedups using even a simple parallelization scheme. With 30 Unix-workstations we have achieved speedups in the range of 20-28.

Future enhancements of the method includes handling of many light sources and simulation of caustics on surfaces with directional diffuse reflection.

7 Acknowledgement

Thanks to my advisor Niels Jørgen Christensen. Thanks to Hans Peter Nielsen for discussing the Museum model with me from an architechtural point of view. Thanks to Greg Ward for discussions related to the simulation of Anisotropic Reflection and directional diffuse reflection.

References

[1] James Arvo, Backward Ray Tracing. *Developments in Ray Tracing. ACM Siggraph Course Notes* **12**, 259-263 (1986).

[2] James Arvo and David Kirk, Particle Transport and Image Synthesis. *Computer Graphics* **24** (4), 53-66 (1990).

[3] Jon Louis Bentley and Jerome H. Friedman, Data Structures for Range Searching. *Computing Surveys* **11** (4), 397-409 (1979).

[4] Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller and Douglas Turner, A Progressive Multi-Pass Method for Global Illumination. Computer Graphics **25** (4), 165-174 (1991).

[5] Cohen, Michael F.; Donald P. Greenberg, The Hemi-Cube - A Radiosity Solution for Complex Environments. *Computer Graphics* **19** (3), p. 31-40, 1985

[6] Steven Collins, Adaptive Splatting for Specular to Diffuse Light Transport. *Eurographics Workshop on Rendering* **5**, (1994).

[7] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg and Benneth Battaile, Modeling the Interaction of Light Between Diffuse Surfaces. Computer Graphics **18**, 213-222 (1984).

[8] E. A. Haines and Donald P. Greenberg, The Light Buffer: A Shadow-Testing Accelerator. *IEEE Computer Graphics and Applications* **6** (9), (1986).

[9] David S. Immel, Michael F. Cohen and Donald P. Greenberg, A Radiosity Method for Non-Diffuse Environments. Computer Graphics **20** (4), 133-142 (1986).

[10] Author name(s) has been given to the Senior Reviewer. "Photon maps in Bidirectional Monte Carlo Ray Tracing of Complex Objects". In Special Scandinavian Section in Computers and Graphics. Vol 19, No. 2, 1995.

[11] James T. Kajiya, The Rendering Equation. Computer Graphics **20** (4), 143-149, 1986

[12] Lafortune, Eric P.; Yves D. Willems: "Bidirectional Path Tracing". *Proceedings of CompuGraphics*, 95-104, june 1993

[13] Nicodemus, F. E.; J. C. Richmond; J. J. Hsia; I. W. Ginsberg; T. Limperis: *"Geometrical Considerations and Nomenclature for Reflectance"*. National Bureau of Standards, okt. 1977

[14] S. N. Pattanaik, Computational Methods for Global Illumination and Visualization of Complex 3D Environments. *Ph.d. Thesis submitted to Birla Institute of Technology & Science* (1993).

[15] Peter Shirley, A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes. *Proc. Graphics Interface*, 205-212 (1990).

[16] Peter Shirley and Changyaw Wang, Luminaire Sampling in Distribution Ray Tracing. *Global Illumination. ACM Siggraph Course Notes* **18** (1992).

[17] François X. Sillion, James R. Arvo, Stephen H. Westin and Donald P. Greenberg, A Global Illumination Solution for General Reflectance Distributions. *Computer Graphics* **25** (4), 187-196 (1991).

[18] John R. Wallace, Michael F. Cohen and Donald P. Greenberg, A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. *Computer Graphics* **21** (4), 311-320 (1987).

[19] Gregory J. Ward, Francis M. Rubinstein and Robert D. Clear, A Ray Tracing Solution for Diffuse Interreflection. *Computer Graphics* **22** (4), 85-92 (1988).

[20] Gregory J. Ward, Adaptive Shadow Testing for Ray Tracing. *Eurographics Workshop on Rendering* **2** (1991).

[21] Gregory J. Ward, Measuring and Modeling Anisotropic Reflection. *Computer Graphics* **26** (2), 265-272 (1992).

[22] Gregory J. Ward and Paul S. Heckbert, Irradiance Gradients. *Third Eurographics Workshop on Rendering.* **3** (1992).

[23] Gregory J. Ward, The RADIANCE Lighting Simulation and Rendering System. *Computer Graphics* **28** (4), 459-472, (1994).

[24] Turner Whitted, An Improved Illumination Model for Computer Graphics. Comm. of the ACM **23** (6), 343-349 (1980).