INTRODUCTION

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

One of the biggest issues in parallel and distributed operating environments is the development of effective techniques for the distribution of the processes of a parallel program on multiple processors. The problem is to schedule the processes among processing elements to achieve some performance goal(s), such as minimizing communication delays and execution time and/or maximizing resource utilization.

Local scheduling performed by the OS of a processor consists of the assignment of the processes to the time-slices of the processor. Global scheduling is the process of deciding where to execute a process in a multiprocessor system. A single authority may carry it out or it may be distributed among processing elements.

The efficient execution of parallel programs depends on the partitioning of the program into modules and schedules these modules for execution on a set of processors. Peak performances are offset by overheads such as:

• Communication overhead.
• Synchronization overhead.
• Loss of efficiency when PE's are out of jobs.
• Operating system task management overhead.

Efficient parallel processing consists of finding a trade-off among the following:

• The number of processors to use.
• Number of modules to execute.
• Amount of overhead.

The above trade-off is achieved using Task Granularity, which is defined as the ratio between task computation time(R) and task communication overhead(C). Fine-grain tasks correspond to small (R/C) and coarse-grain tasks for high (R/C). Since
there is a direct correlation between program efficiency and granularity, the technique for optimization is to cluster a set of fine-grain tasks into coarser-grain partitions.

In this chapter we have given detailed overview of task partitioning and scheduling strategies, which can be used for distributed image computing systems.

4. OVERVIEW OF TASK PARTITIONING

4.1. 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 categorized as shown in Fig 4.1.

![Fig 4.1: Methods of decomposing a problem to exploit parallelism](image)

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 parallelize 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 analyze 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.

4.1.1. Algorithmic decomposition:

In algorithmic decomposition the algorithm itself is analyzed 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 data flow, 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 [2], 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 Fig 4.2.

Sample Problem

```
Begin
  a:=5+6
  b:=(6+4)^2
  c:=a+b
End
```

Fig. 4.2: Algorithmic decomposition: (a) dataflow (b) fork & join
4.1.2. 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:

Step1: 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.

Step2: 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.

4.1.3. 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:

- the *principal data items* (or PDIs) on which the algorithm is to be applied; and
- *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 Fig 4.3.

![Diagram](image)

**Fig. 4.3: A task: the processing of a principal data item**

### 4.1.4. System architecture

This discussion in this section 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 Fig. 4.4. 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 4.4: The System controller as part of a parallel system.
4.2. COMPUTATIONAL MODELS FOR TASK DISTRIBUTION

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 maximize 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 workload 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 illustrates 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 Fig. 4.5, 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 for that position. We will assume that no additional data items are required to complete any task.

Fig. 4.5: Principal data items for calculating the pixels in the image plane.
4.2.1. 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.

**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

\[(\text{number of principal data items}) \mod (\text{number of PEs})\]

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, labeled PE1, PE2 and PE3, as shown in Fig. 4.6. In this case, each processing element is allotted eight principal data items.
As no further principal data item allocation takes place after the initial distribution, a balanced workload 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 workload, 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 Fig. 4.6, 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:
*Solution time* = initial distribution + \( \frac{24}{3} \) + 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:

*Solution time* = initial distribution + \( \frac{24}{5} \) + 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 Fig. 4.7, 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 Fig. 4.8. The computational effort now required to process each of these unequal portions would be approximately the same, minimizing 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 workload 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.

Fig. 4.7 Unequal computational effort due to presence of objects in the scene
The total solution time for a problem using the unbalanced data driven model is thus:

\[ \text{Solution time} = \text{preprocessing} + \text{distribution} + \text{longest portion time} + \text{result collation} \]

So comparing the naive balanced distribution from section 4.2.1

Balanced solution time = distribution + 21 + result collation

Unbalanced solution time = preprocessing + distribution + 14 + result collation

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.

4.2.2. 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 optimize 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 Fig. 4.9 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.

Fig. 4.9: A demand driven model for a simple ray (Image) 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.

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 Fig. 4.8, and Fig. 4.10 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.

If no insight is possible then a simple serial allocation, as shown in Fig. 4.10, or spiral allocation, as shown in Fig. 4.11 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.
Fig. 4.10: Allocation of principal data items using a demand driven model.

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.

Fig. 4.11: Allocation of principal data items in a spiral manner.
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."

### 4.2.3. 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.

### 4.3. 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.
4.3.1. Task definition and granularity

An \textit{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 4.1.2, 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 \textit{task granularity} (or \textit{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 \textit{finest granularity}, while a task, which contains many, is \textit{coarser grained}, or has a \textit{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 \textit{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 would 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 tracing 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.

To summarize 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 color;
- ADI: the scene data; and,
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 color 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 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 synchronization points within the parallel solution which can have a significant effect on overall system performance.

4.3.2. 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, there 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.

4.3.3. 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. Amdahl's law 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:

- *Synchronization 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, utilizing 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 Fig. 4.12. 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 these case elements of the simultaneous equations, become the principal data for the tasks of the next stage.

Even a single stage of a problem may contain a number of distinct sub stages, which must first be completed before the next sub stage can proceed. An example of this is the use of an iterative solver, such as the Jacobi method [3, 4], 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, \( Ax = b \). 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:

![Fig. 4.12: The introduction of partial results due to algorithmic dependencies.](image-url)
Fig. 4.13: 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 4.13. Now PE1 can solve the $n^{th}$ iteration values $x_1^n$, $x_2^n$, and $x_3^n$ in parallel with PE2 computing the values of $x_4^n$, $x_5^n$, and $x_6^n$. However, neither processing element can proceed onto the $(n+1)^{th}$ 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 sub stage, which must be completed before the next sub stage can commence. The following code segment from PE1 illustrates this point:

```
PROCEDURE Jacobi() (* Executing on PE1 *)
Begin
Estimate $x[1] ... x[6]$
$n := 0 (* Iteration number *)
WHILE solution_not_converged DO
Begin
$n := n + 1$
PARALLEL
End
End (* Jacobi *)
```
Data dependencies

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 been 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:

\[
x_1^n = \frac{b_1 - a_{12}x_2^{n-1} - \ldots - a_{16}x_6^{n-1}}{a_{11}}
\]

But the \( x \) value would now be calculated by:

\[
x_2^n = \frac{b_1 - a_{21}x_1^n - a_{23}x_3^{n-1} - \ldots - a_{26}x_6^{n-1}}{a_{22}}
\]

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:

$$x_4^n = \frac{b_4 - a_{14}x_1^{n-1} - a_{15}x_5^{n-1} - a_{16}x_6^{n-1}}{a_{44}}$$

$$x_5^n = \frac{b_5 - a_{15}x_5^{n-1} - a_{25}x_2^{n-1} - a_{16}x_6^{n-1}}{a_{55}}$$

$$x_6^n = \frac{b_6 - a_{16}x_6^{n-1} - a_{26}x_2^{n-1} - a_{15}x_5^{n-1}}{a_{66}}$$

### 4.4. TASK SCHEDULING STRATEGIES

#### 4.4.1. 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 4.2.1. 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:

```plaintext
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
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 pre-fetch the missing principal data items as computation proceeds and local storage allows.

#### 4.4.2. 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 Fig. 4.14.
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 4.2.3. 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.

![Diagram of task supply and demand](image)

**Fig. 4.14:** Supplying task packets from a task pool at the system controller

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 elements 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 realization 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 realization of such a model, often referred to in the literature, is that implemented by May and Shepherd [5]. 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 Fig. 4.15.

![Diagram of a processing element for the processor farm model.](image)

**Key:**

- SC: System Controller
- AP: Application Process
- TR: Task Router
- RR: Result Router

Fig. 4.15: A processing element for the processor farm model.

The application process performs the desired computation, while two router processes, the Task Router (TR) and the Result Router (RR), deal with the communication within the system. 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 synchronized 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:

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

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 initializes the processor farm. Fig. 4.16 shows the manner in which task requests are satisfied within a simple two processing element configured in a chain. The simplicity of this realization 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.

![Diagram of task movement within a two PE processor farm](image)

**Fig. 4.16: Task movement within a two PE processor farm**
4.4.3. 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 have 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 4.14. 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 an 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 4.4.5, 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.

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 Fig. 4.17. 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.

84
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 minimizing communication densities.

4.4.4. Distributed task management

One handicap of the centralized task pool system is that all replenishment task requests from the task managers must reach the system controller before the new tasks can be 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 neighbors. In this case, a replenishment request no longer proceeds directly to the system controller, but simply via the appropriate routers to the neighboring task manager. If this neighboring 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 neighboring task managers, but obviously not the same neighbor 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 Fig. 4.18.

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 neighboring 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 role 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.

4.4.5. 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 Fig. 4.19, on completion of the tasks in its own conceptual region, PE₂ may get allocated task number 22 from PE₁'s conceptual region. Preferred bias allocation is sometimes also referred to as conceptual task allocation.
Fig. 4.19: 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. This can greatly facilitate the even distribution of partial results at the end of any stage of a multistage 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.

4.5. CLASSIFICATIONS OF SCHEDULING STRATEGIES

The general scheduling problem has been described a number of times and in a number of different ways in the literature [6], [7], [8] and is usually a restatement of the classical notions of job sequencing [9] in the study of production management [10]. For the purposes of distributed process scheduling, we take a broader view of the scheduling function as a resource management resource. This management resource is basically a mechanism or policies used to efficiently and effectively manage the access to and use of a resource by its various consumers.

Hence, we may view every instance of the scheduling problem as consisting of three main components:

a) Consumer(s).
b) Resource(s).
c) Policy.
Like other management or control problems, observing the effect it has on its environment may best do understanding the functioning of a scheduler. In this case, one can observe the behavior of the scheduler in terms of how the policy affects the resources and consumers. Note that although there is only one policy, the scheduler may be viewed in terms of how it affects either or both resources and consumers. This relationship between the scheduler, policies, consumers, and resources is shown in Fig. 4.20.

![Fig. 4.20 Scheduling System](image)

In light of this description of the scheduling problem there are two properties, which must be considered in evaluating any scheduling system:

- the satisfaction of the consumers with how well the scheduler manages the source in question (performance), and
- the satisfaction of the consumers in terms of how difficult or costly it is to access the management resource itself (efficiency). In other words, the consumers want to be able to quickly and efficiently access the actual resource in question, but do not desire to be hindered by overhead problems associated with using the management function itself.

One by-product of this statement of the general scheduling problem is the unification of two terms in common use in the literature. There is often an implicit distinction between the terms scheduling and allocation. However it can be argued that these are merely alternative formulations of the same problem, with allocation posed in terms of resource allocation (from the resources' point of view), and scheduling viewed from the consumer's point of view. In this sense, allocation and scheduling are merely two terms describing the same general mechanism, but described from different viewpoints.

4.5.1. The Classification Scheme

The usefulness of the four-category taxonomy of computer architecture presented by Flynn [11] has been well demonstrated by the ability to compare systems through their relation to that taxonomy. The goal of the taxonomy given here is to provide a commonly accepted set of terms and to provide a mechanism to allow comparison of past work in the area of distributed scheduling in a qualitative way.
The taxonomy, while discussed and presented in terms of distributed process scheduling, is applicable to a larger set of resources.

4.5.1.1. Hierarchical Classification of task scheduling strategies

The structure of the hierarchical portion of the taxonomy is shown in Fig. 4.21. A discussion of the hierarchical portion then follows.

4.5.1.1.1. Local Versus Global

At the highest level, we may distinguish between local and global scheduling. Local scheduling is involved with the assignment of processes to the time slice of a single processor. Since the area of scheduling on single-processor systems [6], [12] as well as the area of sequencing or job-shop scheduling [9], [13] has been actively studied for a number of years, this taxonomy will focus on global scheduling. Global scheduling is the problem of deciding where to execute a process, and the job of local scheduling is left to the operating system of the processor to which the process is ultimately allocated. This allows the processors in a multiprocessor increased autonomy while reducing the responsibility (and consequently overhead) of the global scheduling mechanism. Note that this does not imply that a single central authority must do global scheduling, but rather, we
view the problems of local and global scheduling as separate issues, and (at least logically) separate mechanisms are at work solving each.

4.5.1.2. Static Versus Dynamic

The next level in the hierarchy (beneath global scheduling) is a choice between static and dynamic scheduling. This choice indicates the time at which the scheduling or assignment decisions are made.

In the case of static scheduling, information regarding a total mix of processes in the system as well as all the independent subtasks involved in a job or task force [14], [15] is assumed to be available by the time the program object modules are linked into load modules. Hence, each executable image in a system has a static assignment to a particular processor, and each time that process image is submitted for execution, it is assigned to that processor. A more relaxed definition of static scheduling may include algorithms that schedule task forces for a particular hardware configuration. Over a period of time, the topology of the system may change, but characteristics describing the task force remain the same. Hence, the scheduler may generate a new assignment of processes to processors to serve as the schedule until the topology changes again.

Note here that the term static scheduling as used in this section has the same meaning as deterministic scheduling in [7] and task scheduling in [16]. These alternative terms will not be used, however, in an attempt to develop a consistent set of terms and taxonomy.

Examples of Static Scheduling

In [17], we see an example of an optimal, enumerative approach to the task assignment problem. The criterion function is defined in terms of optimizing the amount of time a task will require for all inter-process communication and execution, where the tasks submitted by users are assumed to be broken into suitable modules before execution. The cost function is called a minimax criterion since it is intended to minimize the maximum execution and communication time required by any single processor involved in the assignment. Graphs are then used to represent the module to processor assignments and the assignments are then transformed to a type of graph matching known as weak homeomorphisms. The optimal search of this solution space can then be done using A* algorithm from artificial intelligence [18]. The solution also achieves a certain degree of processor load balancing as well.

Reference [19] gives a good demonstration of the usefulness of the taxonomy in that the paper describes the algorithm given as a solution to the optimal dynamic assignment problem for a two-processor system. However, in attempting to make an objective comparison of this with other dynamic systems, we see that the algorithm proposed is actually static one. In terms of the taxonomy of Section 4.5.1, we would categorize this as static, optimal, graph theoretical approach in which the a priori assumptions are expanded to include more information about the set of tasks to be executed. The way in which reassignment of tasks is performed during process execution is decided upon before any of the
program modules begin execution. Instead of making reassignment decisions during execution, the stronger assumption is simply made that all information about the dynamic needs of a collection of program modules is available a priori. This assumption says that if a collection of modules possess a certain communication pattern at the beginning of their execution, and this pattern is completely predictable, that this pattern may change over the course of execution and that these variations are predictable as well. Costs of reallocation are also assumed to be available, and this assumption appears to be quite reasonable.

The model presented in [20] represents an example of an optimum mathematical programming formulation employing a branch and bound technique to search the solution space. The goals of the solution are to minimize inter-processor communications, balance the utilization of all processors, and satisfy all other engineering application requirements. The model given defines a cost function, which includes inter-processor communication costs and processor execution costs. The assignment is then represented by a set of zero-one variables, and the total execution cost is then represented by a summation of all costs incurred in the assignment. In addition to the above, the problem is subject to constraints, which allow the solution to satisfy the load balancing, and engineering application requirements. The algorithm then used to search the solution space (consisting of all potential assignments) is derived from the basic branch and bound technique.

Again, in [21], we see an example of the use of the taxonomy in comparing the proposed system to other approaches. The title of the paper 'Load Balancing in Distributed Systems' indicates that the goal of the solution is to balance the load among the processors in the system in one way. However, the solution actually fits into the static, optimal, queuing theoretical class. The goal of the solution is to minimize the execution time of the entire program to maximize performance and the algorithm is derived from results in Markov decision theory. An interesting approximate mathematical programming solution, motivated from the viewpoint of fault-tolerance, is presented in [22]. The algorithm is suggested by the computational complexity of the optimal solution to the same problem. In the basic solution to a mathematical programming problem, the state space is both implicitly or explicitly enumerated and searched. One approximation method mentioned in this section [23] involves first removing the integer constraint, solving the continuous optimization problem, discretizing the continuous solution, and obtaining a bound on the discretization error. Whereas this bound is with respect to the continuous optimum, the algorithm proposed in this paper directly uses an approximation to solve the discrete problem and bound its performance with respect to the discrete optimum.

The last static example to be given here appears in [24]. This paper gives a heuristic-based approach to the problem by using extractable data and synchronization requirements of the different subtasks. The three primary heuristics used are:

1) Loss of parallelism,
2) Synchronization,
3) Data sources.
The way in which loss of parallelism is used is to assign tasks to nodes one at a time in order to affect the least loss of parallelism based on the number of units required for execution by the task currently under consideration. The synchronization constraints are phrased in terms of firing conditions, which are used to describe precedence relationships between subtasks. Finally, data source information is used in much the same way a functional program uses precedence relations between parallel portions of a computation, which take the roles of varying classes of suppliers of variables to other subtasks. The final heuristic algorithm involves weighting each of the previous heuristics, and combining them. A distinguishing feature of the algorithm is its use of a greedy approach to find a solution, when at the time decisions are made; there can be no guarantee that a decision is optimal. Hence, an optimal solution would more carefully search the solution space using a back track or branch and bound method, as well as using exact optimization criterion instead of the heuristics suggested.

Examples of Dynamic Scheduling

Among the dynamic solutions presented in the literature, the majority fit into the general category of physically distributed, cooperative, suboptimal, and heuristic. There are, however, examples for some of the other classes.

First, in the category of physically non-distributed, one of the best examples is the experimental system developed for the Cm* architecture-Medusa [15]. In this system, the functions of the operating system (e.g., file system, scheduler) are physically partitioned and placed at different places in the system. Hence, the scheduling function is placed at a particular place and is accessed by all users at that location.

Another rare example exists in the physically distributed non-cooperative class. In this example [25], random level-order scheduling is employed at all nodes independently in a tightly coupled MIMD machine. Hence, the overhead involved in this algorithm is minimized since no information need be exchanged to make random decisions. The mechanism suggested is thought to work best in moderate to heavily loaded systems since in these cases, a random policy is thought to give a reasonably balanced load on all processors. In contrast to a cooperative solution, this algorithm does not detect or try to avoid system overloading by sharing loading information among processors, but makes the assumption that it will be under heavy load most of the time and bases all of its decisions on that assumption. Clearly, here, the processors are not necessarily concerned with the utilization of their own resources, but neither are they concerned with the effect their individual decisions will have on the other processors in the system.

It should be pointed out that although the above two algorithms (and many others) are given in terms relating to general-purpose distributed processing systems, that they do not strictly adhere to the definition of distributed data processing system as given in [26].

In [27], another rare example exists in the form of a physically distributed, cooperative, optimal solution in a dynamic environment. The solution is given for
the two-processor case in which critical load factors are calculated prior to program execution. The method employed is to use a graph theoretical approach to solving for load factors for each process on each processor. These load factors are then used at run time to determine when a task could run better if placed on the other processor. The final class (and largest in terms of amount of existing work) is the class of physically distributed, cooperative, suboptimal, heuristic solutions.

In [28] a solution is given which is adaptive, load balancing, and makes one-time assignments of jobs to processors. No a priori assumptions are made about the characteristics of the jobs to be scheduled. One major restriction of these algorithms is the fact that they only consider assignment of jobs to processors and once a job becomes an active process, no reassignment of processes is considered regardless of the possible benefit. This is very defensible, though, if the overhead involved in moving a process is very high (which may be the case in many circumstances). Whereas this solution cannot exactly be considered as a bidding approach, exchange of information occurs between processes in order for the algorithms to function. The first algorithm (a copy of which resides at each host) compares its own busyness with its estimate of the busyness of the least busy host. If the difference exceeds the bias (or threshold) designated at the current time, one job is moved from the job queue of the busier host to the less busy one. The second algorithm allows each host to compare itself with all other hosts and involves two biases. If the difference exceeds bias1 but not bias2 then one job is moved. If the difference exceeds bias2, then two jobs are moved. There is also an upper limit set on the number of jobs, which can move, at once in the entire system. The third algorithm is the same as algorithm one except that an anti-thrashing mechanism is added to account for the fact that a delay is present between the time a decision is made to move a job, and the time it arrives at the destination. All three algorithms had an adaptive feature added which would turn off all parts of the respective algorithm except the monitoring of load when system load was below a particular minimum threshold. This had the effect of stopping processor thrashing whenever it was practically impossible to balance the system load due to lack of work to balance. In the high load case, the algorithm was turned off to reduce extraneous overhead when the algorithm could not affect any improvement in the system under any redistribution of jobs. This last feature also supports the notion in the non-cooperative example given earlier that the load is usually automatically balanced as a side effect of heavy loading.

The work reported in [29] is an example of an algorithm, which employs the heuristic of load balancing, and probabilistically estimates the remaining processing times of processes in the system. The remaining processing time for a process was estimated by one of the following methods:

- memoryless: \( R(t) = E\{S\} \)
- pastrepeats: \( R(t) = t \)
- distribution: \( R(t) = E\{S - t : S > t\} \)
- optimal: \( R(t) = R(t) \)

where \( R(t) \) is the remaining time needed given that \( t \) seconds have already elapsed, \( S \) is the service time random variable, and \( R(t) \) is the scheduler's estimate of \( R(t) \).
The algorithm then basically uses the first three methods to predict response times in order to obtain an expected delay measure which in turn is used by pairs of processors to balance their load on a pair wise basis: This mechanism is adopted by all pairs on a dynamic basis to balance the system load.

Another adaptive algorithm is discussed in [30] and is based on the bidding concept. The heuristic mentioned here utilizes prior information concerning the known characteristics of processes such as resource requirements, process priority, special resource needs, precedence constraints, and the need for clustering and distributed groups. The basic algorithm periodically evaluates each process at a current node to decide whether to transmit bid requests for a particular process. The bid requests include information needed for contractor nodes to make decisions regarding how well they may be able to execute the process in question. The manager receives bids compares them to the local evaluation and will transfer the process if the difference between the best bid and the local estimate is above a certain threshold. The key to the algorithm is the formulation of a function to be use in a modified McCulloch-Pitts neuron. The neuron (implemented as a subroutine) evaluates the current performance of individual processes. The adaptive nature of this algorithm is in the fact that it dynamically modifies the number of hops that a bid request is allowed to travel depending on current conditions. The most significant result was that the information regarding process clustering and distributed groups seems to have had little impact on the overall performance of the system.

The final example to be discussed here [31] is based on a heuristic derived from the area of Bayesian decision theory [32]. The algorithm uses no a priori knowledge regarding task characteristics, and is dynamic in the sense that the probability distributions, which allow maximizing decisions to be made based on the most likely current state of nature, are updated dynamically. Monitor nodes make observations every $p$ seconds and update probabilities. Every $d$ seconds the scheduler itself is invoked to approximate the current state of nature and make the appropriate maximizing action. It was found that the parameters $p$ and $d$ could be tuned to obtain maximum performance for a minimum cost.

4.5.1.1.3. Optimal Versus Suboptimal

In the case that all information regarding the state of the system as well as the resource needs of a process are known, an optimal assignment can be made based on some criterion function [33], [34], [35], [20], [36], [17]. Examples of optimization measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput. In the event that these problems are computationally infeasible, suboptimal solutions may be tried [22], [37], [38]. Within the realm of suboptimal solutions to the scheduling problem, we may think of two general categories.

4.5.1.1.4. Approximate Versus Heuristic

The first is to use the same formal computational model for the algorithm, but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a "good" one. We will categorize these solutions as
The assumption that a good solution can be recognized may not be so insignificant, but in the cases where a metric is available for evaluating a solution, this technique can be used to decrease the time taken to find an acceptable solution (schedule).

The factors, which determine whether this approach is worthy of pursuit, include:

1. Availability of a function to evaluate a solution.
2. The time required evaluating a solution.
3. The ability to judge according to some metric the value of an optimal solution.
4. Availability of a mechanism for intelligently pruning the solution space.

The second branch beneath the suboptimal category is labeled heuristic [39], [40], [24]. This branch represents the category of static algorithms, which make the most realistic assumptions about a priori knowledge concerning process and system loading characteristics. It also represents the solutions to the static scheduling problem, which require the most reasonable amount of time and other system resources to perform their function. The most distinguishing feature of heuristic schedulers is that they make use of special parameters, which affect the system in indirect ways. Often, the parameter being monitored is correlated to system performance in an indirect instead of a direct way, and this alternate parameter is much simpler to monitor or calculate. For example, clustering groups of processes which communicate heavily on the same processor and physically separating processes which would benefit from parallelism [30] directly decreases the overhead involved in passing information between processors, while reducing the interference among processes which may run without synchronization with one another. This result has an impact on the overall service that users receive, but cannot be directly related (in a quantitative way) to system performance as the user sees it. Hence, our intuition, if nothing else, leads us to believe that taking the aforementioned actions when possible will improve system performance. However, we may not be able to prove that a first-order relationship between the mechanism employed and the desired result exists.

4.5.1.1.5. Optimal and Suboptimal Approximate Techniques

Regardless of whether a static solution is optimal or suboptimal-approximate, there are four basic categories of task allocation algorithms, which can be used to arrive at an assignment of processes to processors.

1. Solution space enumeration and search [17].
2. Graph theoretic [19], [27], [41].
3. Mathematical programming [33], [34], [35], [20], [36].
4. Queuing theoretic [21], [42], [43].

4.5.1.1.6. Dynamic Scheduling

In the dynamic scheduling problem, the more realistic assumption is made that very little, a priori knowledge is available about the resource needs of a process. It is also unknown in what environment the process will execute during its
lifetime. In the static case, a decision is made for a process image before it is ever executed, while in the dynamic case no decision is made until a process begins its life in the dynamic environment of the system. Since it is the responsibility of the running system to decide where a process is to execute, it is only natural to next ask where the decision itself is to be made.

The terms dynamic scheduling and adaptive scheduling are quite often attached to various proposed algorithms in the literature, but there appears to be some confusion as to the actual difference between these two concepts. The more common property to find in a scheduler (or resource management subsystems) is the dynamic property. In a dynamic situation, the scheduler takes into account the current state of affairs, as it perceives them in the system. This is done during the normal operation of the system under a dynamic and unpredictable load. In an adaptive system, the scheduling policy itself reflects changes in its environment-the running system. Notice that the difference here is one of level in the hierarchical solution to the scheduling problem. Whereas a dynamic solution takes environmental inputs into account when making its decisions, an adaptive solution takes environment stimuli into account to modify the scheduling policy itself.

4.5.1.1.7. Distributed Versus Non-distributed

The next issue (beneath dynamic solutions) involves whether the responsibility for the task of global dynamic scheduling, should physically reside in a single processor [15] (physically non-distributed) or whether, the work involved in making decisions should be physically distributed among the processors [26]. Here the concern is with the logical authority of the decision-making process.

When considering the decision-making policy of a scheduling system, there are two fundamental components - responsibility and authority. When responsibility for making and carrying out policy decisions is shared among the entities in a distributed system, we say that the scheduler is distributed. When authority is distributed to the entities of a resource management system, we call this decentralized. This differentiation exists in many other organizational structures. Any system, which possesses decentralized authority, must have distributed responsibility but it is possible to allocate responsibility for gathering information and carrying out policy decisions, without giving the authority to change past or make future decisions.

4.5.1.1.8. Cooperative Versus Non-cooperative

Within the realm of distributed dynamic global scheduling, we may also distinguish between those mechanisms, which involve cooperation between the distributed components (cooperative), and those in which the individual processors make decisions independent of the actions of the other processors (non-cooperative). The question here is one of the degrees of autonomy, which each processor has in determining how its own resources should be used. In the non-cooperative case individual processors act alone as autonomous entities and arrive at decisions regarding the use of their resources independent of the effect of their decisions on the rest of the system. In the cooperative case each processor has the responsibility to carry out its own portion of the scheduling task, but all processors
are working toward a common system-wide goal. In other words, each processor’s local operating system is concerned with making decisions in concert with the other processors in the system in order to achieve some global goal, instead of making decisions based on the way in which the decision will affect local performance only. As in the static case, the taxonomy tree has reached a point where we may consider optimal, suboptimal-approximate, and suboptimal-heuristic solutions. The same discussion as was presented for the static case applies here as well.

In addition to the hierarchical portion of the taxonomy already discussed, there are a number of other distinguishing characteristics which scheduling systems may have. The following sections will deal with characteristics, which do not fit uniquely under any particular branch of the tree-structured taxonomy given thus far, but are still important in the way that they describe the behavior of a scheduler. In other words, the following could be branches beneath several of the leaves shown in Fig. 4.2.1 and in the interest of clarity are not repeated under each leaf, but are presented here as a flat extension to the I scheme presented thus far. It should be noted that these attributes represent a set of characteristics, and any particular scheduling subsystem may possess some subset of this set. Finally, the placement of these characteristics near the bottom of the tree is not intended to be an indication of their relative importance or any other relation to other categories of the hierarchical portion. Their position was determined primarily to reduce the size of the description of the taxonomy.

4.5.1.2. Flat Classification Task Scheduling Strategies

4.5.1.2.1. Adaptive Versus Non-adaptive

An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to previous and current behavior of the system in response to previous decisions made by the scheduling system. An example of such an adaptive scheduler would be one, which takes many parameters into consideration in making its decisions [30]. In response to the behavior of the system, the scheduler may start to ignore one parameter or reduce the importance of that parameter if it believes that parameter is either providing information, which is inconsistent with the rest of the inputs or is not providing any information regarding the change in system state in relation to the values of the other parameters being observed. A second example of adaptive scheduling would be one which is based on the stochastic learning automata model [44]. An analogy may be drawn here between the notion of an adaptive scheduler and adaptive control [45], although the usefulness of such an analogy for purposes of performance analysis and implementation are questionable [46]. In contrast to an adaptive scheduler, a non-adaptive scheduler would be one, which does not necessarily modify its basic control mechanism on the basis of the history of system activity. An example would be a scheduler, which always weighs its input in the same way regardless of the history of the system’s behavior.
4.5.1.2.2. Load Balancing

This category of policies, which has received a great deal of attention recently [21], [47], [48], [36], [49], [50], [51], [28], approaches the problem with the philosophy that being fair to the hardware resources of the system is good for the users of that system. The basic idea is to attempt to balance (in some sense) the load on all processors in such a way as to allow progress by all processes on all nodes to proceed at approximately the same rate. This solution is most effective when the nodes of a system are homogeneous since this allows all nodes to know a great deal about the structure of the other nodes. Normally, information would be passed about the network periodically or on demand [52], [53] in order to allow all nodes to obtain a local estimate concerning the global state of the system. Then the nodes act together in order to remove work from heavily loaded nodes and place it at lightly loaded nodes. This is a class of solutions, which relies heavily on the assumption that the information at each node is quite accurate in order to prevent processes from endlessly being circulated about the system without making much progress. Another concern here is deciding on the basic unit used to measure the load on individual nodes.

4.5.1.2.3. Bidding

In this case of policy mechanisms, a basic protocol framework exists which describes the way in which processes are assigned to processors. The resulting scheduler is one, which is usually cooperative in the sense that enough information is exchanged (between nodes with tasks to execute and nodes which may be able to execute tasks) so that an assignment of tasks to processors can be made which is beneficial to all nodes in the system as a whole.

To illustrate the basic mechanism of bidding, the framework and terminology of [54] will be used. Each node in the network is responsible for two roles with respect to the bidding process: manager and contractor. The manager represents the task in need of a location to execute, and the contractor represents a node, which is able to do work for other nodes. Note that a single node takes on both these roles, and that there are no nodes, which are strictly managers or contractors alone. The manager announces the existence of a task in need of executing by a task announcement, and then receives bids from the other nodes (contractors). A wide variety of possibilities exist concerning the type and amount of information exchanged in order to make decisions [28], [55]. The amount and type of information exchanged are the major factors in determining the effectiveness and performance of a scheduler employing the notion of bidding. A very important feature of this class of schedulers is that all nodes generally have full autonomy in the sense that the manager ultimately has the power to decide where to send a task from among those nodes which respond with bids. In addition, the contractors are also autonomous since they are never forced to accept work if they do not choose to do so.
4.5.1.2.4. Probabilistic

This classification has existed in scheduling systems for some time [9]. The basic idea for this scheme is motivated by the fact that in many assignment problems the number of permutations of the available work and the number of mappings to processors so large, that in order to analytically examine the entire solution space would require a prohibitive amount of time.

Instead, the idea of randomly (according to some known distribution) choosing some process as the next to assign is used. Repeatedly using this method, a number of different schedules may be generated, and then this set is analyzed to choose the best among those randomly generated. The fact that an important attribute is used to bias the random choosing process would lead one to expect that the schedule would be better than one chosen entirely at random. The argument that this method actually produces a good selection is based on the expectation that enough variation is introduced by the random choosing to allow a good solution to get into randomly chosen set.

Alternative views of probabilistic schedulers are those, which employ the principles of decision theory in the form of team theory [56]. These would be classified as probabilistic since suboptimal decisions are influenced by prior probabilities derived from best guesses to the actual states of nature. In addition, these prior probabilities are used to determine (utilizing some random experiment) the next action (or scheduling decision).

4.5.1.2.5. One-Time Assignment Versus Dynamic Reassignment

In this classification, we consider the entities to be scheduled. If the entities are jobs in the traditional batch processing sense of the term [57], [7], then we consider the single point in time in which a decision is made as to where and when the job is to execute. While this technique technically corresponds to a dynamic approach, it is static in the sense that once a decision is made to place and execute a job; no further decisions are made concerning the job. We would characterize this class as one-time assignments. Notice that in this mechanism, the only information usable by the scheduler to make its decision is the information given it by the user or submitter of the job. This information might include estimated execution time or other system resource demands. One critical point here is the fact that once users of a system understand the underlying scheduling mechanism, they may present false information to the system in order to receive better response. This point fringes on the area of psychological behavior, but human interaction is an important design factor to consider in this case since the behavior of the scheduler itself is trying to mimic a general philosophy. Hence, the interaction of this philosophy with the system's users must be considered.

In contrast, solutions in the dynamic reassignment class try to improve on earlier decisions by using information on smaller computation units-the executing subtasks of jobs or task forces. This category represents the set of systems, which 1) do not trust their users to provide accurate descriptive information, and 2) use dynamically created information to adapt to changing demands of user processes. This adaptation takes the form of migrating processes (including current process
state information). There is clearly a price to be paid in terms of overhead, and this price must be carefully weighed against possible benefits.

An interesting analogy exists between the differentiation made here and the question of preemption versus non-preemption in uniprocessor scheduling systems. Here, the difference lies in whether to move a process from one place to another once an assignment has been made, while in the uniprocessor case the question is whether to remove the running process from the processor once a decision has been made to let it run.
REFERENCES


[57] I. Flores, OSMFT, 1973, Allyn and Bacon, Boston MA.