PARALLEL PROCESS PLACEMENT

THESIS

Submitted in Partial Fulfilment of the

Requirements for the Degree of

MASTER OF SCIENCE (APPLIED COMPUTER SCIENCE)

of Rhodes University

by

Caroline Handler

January 1989

Abstract

This thesis investigates methods of automatic allocation of processes to available processors in a given network configuration.

The research described covers the investigation of various algorithms for optimal process allocation. Among those researched were an algorithm which used a branch and bound technique, an algorithm based on graph theory, and an heuristic algorithm involving cluster analysis. These have been implemented and tested in conjunction with the gathering of performance statistics during program execution, for use in improving subsequent allocations.

The system has been implemented on a network of loosely-coupled microcomputers using multi-port serial communication links to simulate a transputer network. The concurrent programming language occam has been implemented, replacing the explicit process allocation constructs with an automatic placement algorithm. This enables the source code to be completely separated from hardware considerations.

Acknowledgements

The author would like to thank her supervisor, Professor Peter Clayton, for so willingly imparting his knowledge of computing and for his unfailing patience. Special appreciation is expressed for the help received from the two particularly close associates who assisted in the solution of intricate technical and programming problems. The constructive criticism received from colleagues during the preparation of this thesis contributed to the properly proportioned form of the final presentation.

Contents

List of Illustrations

Program listings of the static process allocation algorithms are available in Tec.Doc 89/03, Department of Computer Science, Rhodes University, Grahamstown.

1. Introduction

Advances in hardware technology have improved the computing power of single processors by orders of magnitude in the past decades. As the performance of processing devices approaches the theoretical maximum speed, improvements in performance become more costly and less significant. The linking of several processors together in multiprocessor computing architectures is an economically feasible alternative for significantly increasing the performance of computers, and thereby enabling the computationally intensive requirements of real-time systems to be met.

The main driving force behind the evolution of distributed systems is that they offer a high degree of availability, extensibility, reliability, and concurrency. In order to reap these benefits, applications must be expressed in distributable components (called processes) [HAN78] which may be assigned to different processors before or during execution. A major difference which has emerged between various designs of distributed systems is the location of memory and the format of inter-processor communication. Devices exist with only local memory and all communication is carried out on a point-to-point basis. These systems have been developed to combat the bottlenecks which result from shared memory systems, or systems using buses for communication, due to the limited bandwidth of the shared device.

Recently, much research effort has concentrated on the design of distributed operating systems, but the attention has been focused mainly on issues concerning inter-process communication, synchronization, and the creation of processes [GON80]. The assignment of processes to processors is one of the least understood areas of the design of distributed computing systems. To distribute applications in a satisfactory manner, techniques are required for choosing the best process placements from several possibilities.

Designers of concurrent languages can either provide facilities within the language which require the programmer to specify allocation in the source code, as in *MOD [COO80] and occam [BUR88a], or else can leave allocation to the implementation, as is the case with Ada [BIS87b] and Modula-2 [GOU88]. Since one of the design objectives in distributed operating systems is architectural transparency, the process placement function should ideally be performed by the system.

Repetitive trial-and-error procedures are presently being used to perform workload partitioning for distributed computer systems. One reason why computerized optimization methods have not yet been widely applied to this problem is the lack of established, and at the same time practical, mathematical models of the workload partitioning problem. This thesis attempts to define such a mathematical model and to investigate the computational techniques for solving the resulting optimization problem.

Three process allocation algorithms have been implemented and tested on a network of loosely-coupled[1] microcomputers in an attempt to remove, from the programmer, the burden of process allocation within the source code, and rather to develop a system of automatic process placement. All three algorithms researched and implemented are based on static allocation as opposed to dynamic allocation, and thus some form of statistical gathering is needed to obtain the necessary system and application program characteristics. The execution of an application program becomes an iterative process where statistics are gathered after each run and then used to base a subsequent improved process allocation, with the aim of decreasing the execution speed of the application program.

The process of repetitively gathering performance statistics is itself a time consuming exercise. To achieve cost effectiveness, the use of this approach is directed at large, frequently used applications with very high processing time. Information concerning the distribution of processes for a particular class of program and data set may be gathered and used to direct a user's decision as to which process allocation approach to adopt for his particular application.

2. The Multiprocessor Model

Multiprocessor architectures can be broadly grouped into MIMD and SIMD machines [VAN81]. MIMD machines are Multiple-Instruction-Multiple-Data systems. They contain a number of processors each executing their own programs on their own data sets. Processors are inter-connected to allow programs to exchange data and synchronize activities. MIMD systems are usually implemented as either shared memory machines or message passing machines. Examples of MIMD machines are the CRAY-XMP [SHA87] and transputer. SIMD machines are Single-Instruction-Multiple-Data systems. They contain a number of processors, each of which executes the same program on different data sets. An array processor is an example of a SIMD system.

The system developed during the research described in this thesis has been implemented on a network of loosely-coupled microcomputers using multi-port serial communication links to simulate a transputer network [HIL86]. The topology that was used for the test runs was that of a star network, as shown in Figure 1, with a 4-port serial board being placed in the machine residing at the center of the star, and communications being passed through the serial ports of the other machines in the network. Communication between processors which are not directly linked in the topology is made possible by a rerouting scheme to be discussed in Section 5.1.

1. *Loosely-coupled microcomputer systems are those which are fully distributed and thus have no form of shared memory. The transfer of all information is usually two-party co-operative, in the same way as input and output are performed [SHA87].*

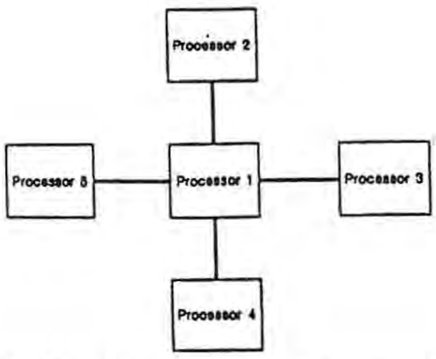


Figure 1 - star topology forming multiprocessor network

Typically, loosely coupled systems provide good throughput at the expense of flexible resource sharing, while tightly coupled systems provide flexible resource sharing at the cost of degraded throughput [CHU80].

The network considered for this research is fully inter-connected and homogeneous in that it consists of several functionally similar processors. In the model, each processor is an independent computer with full memory, control, and arithmetic capability. The processors may differ in their processing power. For example, a network may consist of an arrangement of 8088 and 80286 based microcomputers.

It is envisaged that a microcomputer be assigned the task of gathering all the statistics made available by the other processors in the network and to perform the task of devising an alternative distribution, based on the gathered statistics, using one of the algorithms for process placement to be discussed below.

3. A Transputer Based System

With the emergence of VLSI technology, it has been possible to construct a powerful microcomputer with memory, processor and communications on a single device [MAY88]. An important characteristic of such devices is that they can be easily linked together to form a network whose combined processing power far surpasses that of any uniprocessor system [POU86] [KAR80].

One of the best known devices of this type is the transputer [WAL85]. There are other devices based on the same technology as the transputer, but it seems that it is only the transputer that is beginning to be accepted by computer users outside the academic and research environments. The word transputer refers to a device that integrates a reduced-instruction-set-computer processor, some internal memory (typically 4K), and a set of inter-processor communication links. The T800 transputer design [INM87] features a 32-bit integer processor, a 32-/64-bit IEEE floating point unit, and four full-duplex serial I/O links driven by a dedicated 8-channel direct memory access engine that can sustain 20 mips (millions of instructions per second) on each link. An important characteristic of transputer systems is that computational power increases as processors and memory are added. The transputer is a multicomputer building block, designed to execute concurrent processes under direct hardware control [MUR88].

To construct a multiprocessor system, transputer nodes are configured into a chosen topology. Processes which have been allocated to separate processors may themselves consist of subprocesses. Transputer technology does not differentiate between these two classes of processes as far the generated code is concerned.

To date, the TDS (Transputer Development System), an integrated development system of the transputer, and the working environment for the user, comprises the tools needed to program concurrent systems in occam, the native parallel programming language of the transputer. However, compilers of languages such as Pascal, Modula-2, Lisp, Fortran, and C, as well as new operating systems, e.g. Helios, are emerging [CLA88].

Although occam was designed to be an abstract programming language, it is used predominantly to program transputers, and is often regarded as a high level assembly language of the transputer because of the transputer's facilities for supporting the execution of occam.

4. Process Allocation - the Occam Approach

The design of occam is based on that of CSP (Communicating Sequential Processes) [HUL85], which emerged as a message-based implementation of systems involving communicating processes.

In occam, communicating processes exchange data through a construct known as a channel. An occam channel is a one-way point-to-point pathway resident in memory. Communicating processes are synchronized by channel communication, and the transputer links are memory mapped so that channels can be placed at link addresses. This placement transforms a *soft* channel into a *hard* channel, and data is transferred through a link to a process attached to the link on another transputer [NEW86].

Occam provides the framework for constructing parallel processes. Parallel processes that communicate must do so through occam channels, not via shared memory. This is to prevent deadlock and to facilitate the execution of processes on separate processors which have no memory in common.

The point-to-point nature of occam channels discourages the design of a program dependent on routing data through intermediate nodes, although system software written for the transputer, e.g. the Helios Operating System [GAR87], may cater for this in some way [BRO86].

Occam is designed for concurrent programming on a network of transputers. The partitioning of the application program into its constituent subprocesses is specified **by the programmer within the source code**, as is the subsequent allocation of these subprocesses to available processors in a transputer network. By placing allocation completely in the hands of the programmer, the designers of occam allow applications, written in the language, to be easily implemented on a distributed system.

The design of occam includes a *placed par* construct that enables the user to specify the allocation of processes from within the source code. The *placed par* construct is a variant of the *par* constructor and indicates that the associated subprocesses are not only concurrent, but that they are to be allocated to different processors and will therefore be truly parallel. The model of concurrent processes in occam allows for one or more processes to execute on each processor. Processes communicate via logical channels and, where appropriate, these need to be mapped onto the physical communication links between processors. The syntax of the *placed par* construct and the allocation of channels is shown below [POU88].

```
parallel   =  PLACED PAR
                          {placement}
                    | PLACED PAR replicator
                          {placement}
placement  =  PROCESSOR expression
                          process
allocation =  PLACE channel AT expression
process    =  allocation
                          process
```

```
for example:
  PLACED PAR
    PROCESSOR 1
    PLACE channel AT link0
      p1
    PROCESSOR 2
    PLACE channel AT link2
      p2
```

where *link0* and *link2* are implementation dependent constants.

A transputer network not only needs the above control structures within occam, but also a piece of software, named the *configurer*, which resides independently of the application program, and which specifies the relationship of processes to processors. The use of these control structures within an application program results in a loss of portability, since the source code must be modified if it is to be executed on a transputer network with a different topology. Moreover, specifying allocation in the source code becomes excessively tedious for programming a system of many processes [HIL88].

5. Towards an Alternative Process Allocation Approach

The aspects of occam discussed in the previous section restrict its use as an abstract programming language and Hill [HIL88] proposes an alternative approach which **completely separates the source code from the hardware considerations.** His implementation eliminates the *placed par* construct and introduces an allocation scheme whereby the target transputer network and the allocation scheme are supplied by the user after successful compilation. Ideally, the allocation should be automatic, i.e. produced by the system, and it is to this end that the research described in this paper, is directed.

The language implementation of the proposed amended occam, including the characteristics discussed below of message packets, the matcher protocol, the network map and the allocator, were implemented by Hill [HIL88] and formed the ground work on which this thesis has been based. The remainder of this chapter summarizes those aspects of Hill's work upon which the rest of this thesis is built.

5.1. Communication

The ability to provide global communication is essential if the programmer is to be abstracted from hardware considerations and yet be given the ability to execute the program making efficient use of available hardware [HIL88]. This is necessary so as not to restrict the set of possible allocations of a program, thereby possibly excluding the most efficient allocation.

Provision is made for a virtual link between every pair of transputers in the network. Those pairs which are not directly connected by a physical link, communicate via other transputers which pass on the message. Any number of occam channels must be able to be mapped to a virtual link. In order to facilitate this, all messages are sent in the form of packets. The structure of a message packet is shown in Figure 2. Of interest is the message packet routing information. On receiving a message packet, the processor examines the first number of the packet. A zero indicates that the message is destined for that processor. If the first number of the message packet is non-zero, it indicates the port through which the rest of the message must be forwarded.

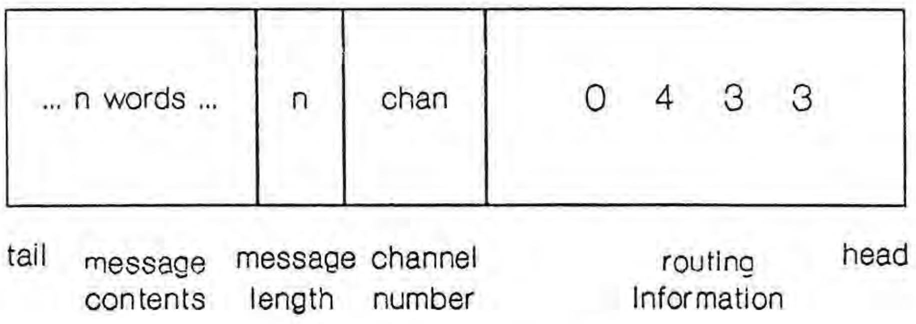


Figure 2 - Message Packet[2]

Channel to virtual link mappings may change dynamically through the course of the execution of the program. To overcome this, a third-party channel supervisor, known as the *matcher*, has the function of monitoring the status of the channel and coordinating synchronization. A separate matcher exists for each instance of a channel and may exist on a processor which does not execute either the sender or the receiver process. The matcher resides on the processor which houses the process that spawned the two communicating daughter processes.

2. *Reprinted from [HIL88].*

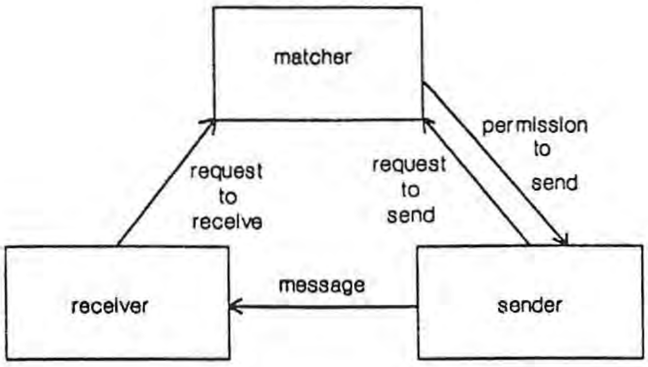
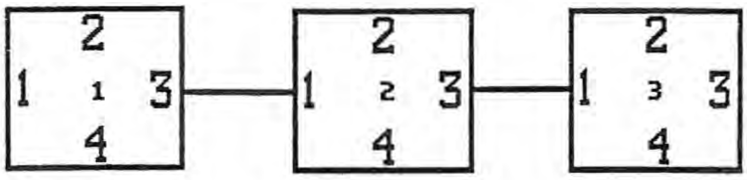


Figure 3 - Protocol using matcher[3]

By allowing communication between any two transputers in the network, the potential communication overhead in forwarding messages increases exponentially for each transputer that is added to the network. It is therefore essential that the allocation algorithm used should attempt to minimize this overhead and the allocation should reflect a clustering of processes which communicate with each other.

5.2. The Network Map

The *Network Map* provides the mapping of virtual *links to* physical links and allows for a communication layer to be implemented on each transputer so as to abstract the logical connections between processes from the physical connections between transputers. Figure 4 shows an example of such a description for a sequence of three processors.



```
3     {there are 3 processors in the network}
0     {processor 1 to processor 1 link}
3 0   {processor 1 to processor 2 link}
3 3 0 {processor 1 to processor 3 link}
1 0   {processor 2 to processor 1 link}
0     {processor 2 to processor 2 link}
3 0   {processor 2 to processor 3 link}
1 1 0 {processor 3 to processor 1 link}
1 0   {processor 3 to processor 2 link}
0     {processor 3 to processor 3 link}
```

Figure 4 - an example of a network map[3]

3. *Reprinted from [HIL88].*

The *Network Map* interface of the existing system requires further research and development to make it more user-friendly, or in fact, to make it an automatic function performed by the system. This area of expansion does not fall within the scope of this project.

5.3. The Allocator

A piece of software, known as the *allocator* (refer to Figure 7), makes use of the description of the target transputer network in order to map daughter processes to transputers.

```
3 {there are 3 processes to be allocated}
1 {process 1 is allocated to processor 1}
2 {process 2 is allocated to processor 2}
3 {process 3 is allocated to processor 3}
```

Figure 5 - a simple allocation scheme for three processes[4]

The system proposed by Hill facilitates automatic configuration by omitting process allocation details from the source code, and transferring this responsibility to the implementation. It is proposed that every daughter process can be given the potential to execute on any transputer in the target network as constrained by the allocation algorithm employed. This is illustrated by the example program in Figure 6, amended from [HIL88], where the blocked code may execute on any transputer, which is not necessarily the transputer executing the code contained in the outer block. The purpose of this example is to illustrate the problem of globally declared variables when attempting to distribute processes.

```
VAR x,y,z:
SEQ
  x := 4
  PAR
    VAR a:
    SEQ
      a := 1
      IF
        x = 4
          PAR
            SEQ
              -- a process which alters a
            SEQ
              -- a process which alters x
        TRUE
          SEQ
            -- a process which alters x
    SEQ
      -- a process which makes use of y and z
```

Figure 6 - The Proposed Partitioning

4. *Reprinted from [HIL88].*

A schematic representation of the proposed system is given in Figure 7.

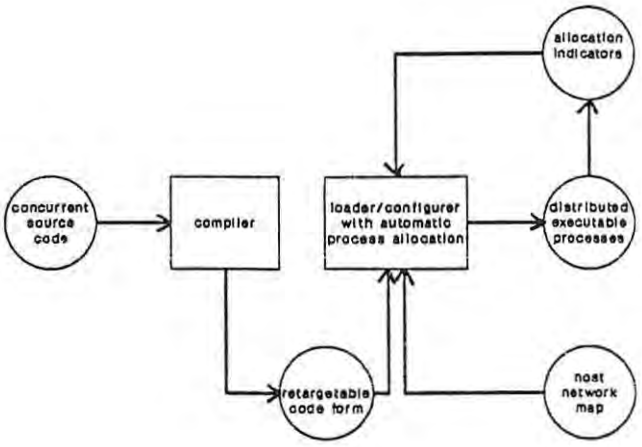


Figure 7 - the abstracted occam system.

In Hill's implementation the allocation of processes is not automatic; it is provided by the user at compile time. It has been the aim of this research work to replace this with automatic process placement.

6. The Theory of Process Allocation

The assignment of processes to processors affects system response time, throughput and reliability. The goal in process placement is to balance the processing load among the processors such that either the total system time is minimized, or processor loads are balanced.

Factors which have to be taken into consideration when deciding upon a particular allocation scheme are:

1. the number of processes to be allocated versus the number of processors available in the network,
2. the computing power of each processor,
3. the workload created by each process, and
4. the inter-dependency between processes, e.g. a high level of communication between two processes warrants that they be placed on the same processor. If communicating processes are placed on different processors, then the sending processor needs to spend time formatting messages and initialising addresses, while the receiving processor spends time on extracting the message contents and notifying the destination process.

A program is partitioned into functional modules, some of which are assigned to particular processors, the remainder of which are permitted to "float" from processor to processor during program execution. Some modules have a fixed assignment because these modules depend on facilities within their assigned processor. The facilities might be a high-speed arithmetic capability, access to a particular database, the need for a large high-speed memory, access to a specific peripheral device, the use of a fast floating-point unit, or the need for any other facility that might be associated with some processor and not with every processor.

Some processors have unique capabilities in the network, although all are functionally similar in most applications. When a process is submitted to a network, it may have to be assigned to a certain fixed processor in order to take advantage of its unique capabilities. For this reason, some developers of allocation algorithms classify processes that need to be allocated into the following three categories [EFE82]:

- processes that can only be assigned to particular processors,
- processes that can only be assigned to a limited set of processors, and
- processes that can be assigned to any processor in the network.

Processes are allocated among distributed processors to achieve the following goals [ARO81]:

- to allow the specification of a large number of constraints,
- to balance the utilization of individual processors in the distributed computing system,
- to minimize inter-processor communication costs,
- to take into account a system of various processors having differing capabilities, memory sizes and communication links, and
- the algorithm must be efficient in that it must exhibit a great improvement on the manual allocation scheme and, more importantly, the efficiency must be such that timing constraints are met, as failure to do so in a real-time application, results in incorrect program execution. Because most of the information used by the algorithm is an estimation regarding the run-time behaviour, it is doubtful whether satisfying such timing constraints can be guaranteed.

In order for the process placement model to satisfy these goals, the following two supporting functions should be present:

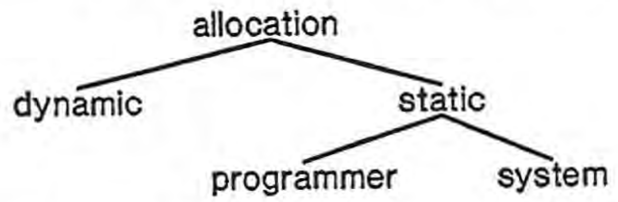
1. The *process preprocessor* analyses the application processes to acquire relevant information such as coupling factors among processes and process attribute sizes. The coupling factor is an initial estimate of the number of data units transferred from one process to another. The process attributes are the inherent characteristics of processes, e.g. the number of executable statements and the maximum allowable execution time.

2. The *network preprocessor* examines the distributed network and provides information on the network architecture[5], e.g. inter-processor distance and processor (hardware) constraints.

The extraction of the information needed by the above preprocessors is a non-trivial exercise. As a result, human intervention is needed to input some of the data which is not easily accessible. Consequently, the NETWORK MAP, described in Section 5, entirely replaces the network preprocessor.

5. *The Helios Operating System includes what is known as a 'worm' which circulates through a network of processors, gathering information as it goes along [GAR87].*

Allocation can be performed either statically or dynamically [LIU73].



In dynamic allocation, the allocation occurs at run-time with the advantage of better processor utilization since the allocation is in response to the run-time state of the processors [HIL88]. The disadvantage is the processing time used each time an allocation decision is made, as well as the resulting scheduling required to implement the altered allocation, i.e. allocation costs are paid each time the program is executed. Gajski [GAJ85] surveys various implementations of centralized and decentralized dynamic allocation algorithms.

Static allocation involves mapping processes to processors before the execution of the program [HIL88]. The *placed par* construct in occam enables the programmer to perform the allocation. Alternatively, Bishop *et al* [BIS87b] and Mellor *et al* [MEL86] suggest ways of performing the mapping automatically within the system, before execution. The advantage of the static allocation approach is that allocation costs are only paid once for each allocation made. The disadvantage is that to obtain an improved allocation of processes, it is necessary to predict, at compile time, the run-time behaviour of the application program. This requires the use of specialised software for which algorithms exist for only a limited group of common problems, e.g. sorting. Because of this, less efficient allocations might be adopted.

One of the major problems preventing widespread use of distributed systems is the difficulty in verifying the effectiveness of allocations resulting from an allocation model. Due to the lack of real-life data, most research results are limited to theoretical and mathematical models, and their merits are difficult to compare [MAP82].

The static allocation option has been adopted for the research described below since it is more suited to transputer networks; dynamic allocation requires global communication which is expensive in a transputer network. Three approaches to the allocation model have been identified; they are integer programming, graph theoretical, and heuristic methods.

6.1. The Use of Statistics in Static Process Allocation

A disadvantage of static allocation is that it is difficult to predict, at compile time, how often a channel will be used or how much processor time a process will require. This information is obviously important to any allocation algorithm since the algorithm must minimize communication overheads while maintaining an even load balance among the transputers. Therefore, provision is made for recording run-time statistics of processor and channel usage by each daughter process. The allocation is then viewed as an iterative process, with the program being configured, executed, reconfigured and re-executed until the user is satisfied with the allocation. The idea of using run-time statistics in order to aid the allocation process has been used by [FIS86], [MEL86], [LAM84] and [RAT87].

The cost of running a process on each processor must be known in order to compute an improved placement, and it is certainly infeasible to ship a process to every other processor in the system for a time trial to determine its relative running time. Stone [STO77] suggests that a reasonable approximation is to assume that the running time of a process on processor P_1 is a fixed constant times the running time of that process on processor P_2, where the constant is determined in advance as a measure of the relative power of the processors without taking into consideration the precise nature of the program to be executed. Under these assumptions, only data about inter-process communication need be gathered.

The success of the use of statistics in determining efficient process allocations is highlighted by some of the performance graphs given in the respective sections discussing each allocation algorithm implemented. These graphs show a sharp drop in program execution time between the initial allocation, and those after some statistics have been gathered. It is interesting to note that an improvement can be seen on the graph after very few runs, and subsequent runs do not improve the allocation significantly. It should also be noted that although the initial allocation is the worst case allocation for many classes of application programs, there is a possibility that, if the grain of parallelism is small, it may be the best.

Test runs performed during the implementation of a static allocation approach show that the gathering of statistics does increase the overall execution time of a particular application program by about 10%. The statistics gathered included the recording of the number of pcode steps executed by each process, the execution time of each process (using the real-time hardware clock), and the proportion of execution time spent by each process on communication. Factors such as the amount of time a process was suspended due to communication or timeslicing were also considered. The expanded version of the system, developed as part of this project, gave the user the choice of suppressing the gathering of statistics.

The run-time behaviour of a program usually depends on the input data it is given [HIL88]. This means that statistics gathered from the repeated execution of the same program may differ according to the input data used for each run. Hill [HIL88] suggests that it might be possible to apply cumulative statistics, averaged over a wide range of input data sets, to a class of programs and in this way "fine-tune" a configuration for improved execution of a particular application program.

The choice of exactly what run-time statistics should be calculated is dependent on the requirements of the allocation algorithm in use. The following sections address some of the properties of statistical gathering and static process allocation algorithms are discussed.

6.2. The Cost Function

There are two major costs that must be considered in an assignment of processes. In order to reduce the cost of *inter-processor communication*, the program modules in a working set should be co-resident in a single processor during execution of that working set. To reduce the *computational cost* of a program, program modules should be assigned to processors on which they run fastest. The two kinds of assignments can be incompatible. The problem is to find an assignment of program modules to processors that minimizes the collective costs due to inter-processor communication and computation.

The usual method of measuring CPU (Central Processing Unit) power is in terms of throughput, usually mips. When measuring the processing power of a multiprocessor, a ratio is more accurate. That is, a system with n processors yields x times the throughput of a system with m processors. There are two methods for measuring throughput: system throughput and CPU throughput capacity [CHU87d]. The system throughput ratio of a system a to system b is defined as

$$\begin{aligned} R_{ab} &= \frac{\text{throughput}_a}{\text{throughput}_b} \\ &= \frac{X/T_a}{X/T_b} \qquad\qquad \text{where ,} \\ &= \frac{T_b}{T_a} \qquad\qquad\qquad X = \text{total work} \\ & \qquad\qquad\qquad\qquad T_a = \text{elapsed time of system a.} \end{aligned}$$

When only CPU power is being compared, the system throughput is not an accurate measure. This is because, as the computational power increases, the non-CPU related components of the workload, e.g. I/O, occupy a larger fraction of the total elapsed time. CPU throughput capacity is the maximum service rate that the CPU exhibits during a measured run of the amount of work done per unit of CPU time.

Chu, et al [CHU80] suggest a method of estimating inter-process communication. Processes and inter-process communication are represented by a flow graph, an example of which is given in Figure 8, and a set of equations are solved which estimate the inter-process communication using branching probabilities and loop frequencies. Inter-process communication is classified into three categories of 1) shared or intermediate data transfer, e.g., the transfer of information between processes that are not coresident, 2) distributed database management, and 3) system control, e.g., process scheduling and synchronization.

Figure 8 is a control and data flow graph of the partitioning of a program into its constituent processes. The rectangular blocks are processes, and the directed arcs represent precedence relationships between connected processes. The arcs between processes imply data transfer from one process to another. Communication between process 2 and process 8 (in the diagram) is an example of type 2 communication where there is no physical link between the two communicating processes.

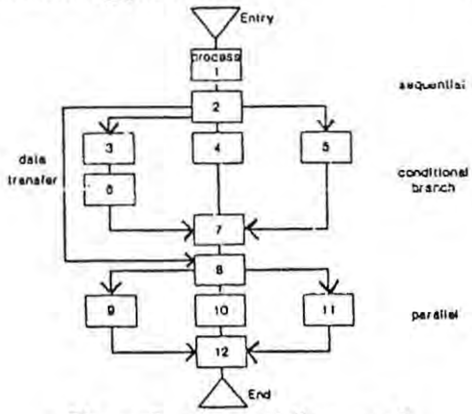


Figure 8 - a control flow graph

Three basic types of control flow are identified: sequential, conditional branching (exclusive OR), and parallel (AND) branching. In the following equations it must be assumed that the probability of each arc (in the graph) being chosen is known, as well as the average number of times each loop is executed.

If $N^{a_{ij}}$ = the number of times the arc i-j is traversed, and
N^{m_i} = the number of times a module M_i is executed

then the inter-process communication can be calculated according to the following formula :

$$v_{ij} = \begin{cases} v'_{ij} \times N^{a_{ij}} & \ldots\text{for type-1 communication} \\ v''_{ij} \times N^{m_i} & \ldots\text{for type-2 communication} \\ 0 & \ldots\text{otherwise,} \end{cases}$$

where ,

v'_{ij} = the volume of the communication each time the arc i-j is traversed, and
v''_{ij} = the volume of communication which occurs between processes which are not t, e.g. coresident, e.g. the arc i-j joining process 2 and process 8. The rules for etail in determining $N^{a_{ij}}$ are listed in detail in [CHU80]. The values of v'_{ij} and v''_{ij} of the are said to be identical and equal to the intersection of the output from process M_i and the input to process M_j, respectively.

Inter-process communication is also involved when a process residing on one processor needs to access a file on another processor. In this instance,

$v_{ij} = \sum f_{ik} d_{ikj}$

where f_{ik} = an estimate of the frequency with which the kth I/O statement in process M_i accesses file j, and
d_{ikj} = the volume of communication invoked by executing the kth I/O statement in process M_i .

Chu et al [CHU87b] introduce a model that estimates process response time, a factor which is identified as being important in the process placement problem. These types of estimations are very important in the implementation of heuristic process allocation algorithms, as will be seen in Chapters 9 and 11. Chu uses queueing networks as a means for modeling distributed processing systems in an attempt to estimate all constituent costs.

The cost function used for the research is an m*m matrix, where m is the number of processes to be allocated. For each possible combination of process communication, two costs are assigned:

COST 1 = cost if two communicating processes i and j, are placed on the same processor,

COST 2 = cost if two communicating processes i and j, are placed on different processors.

The matrix is initialised to all zeroes and those processes which never communicate, remain with a cost of zero.

The costs which are entered into the matrix, are obtained from results of run-time statistics. The first run is regarded as the initial allocation which is an intelligent guess by the user, with all processes being allocated to the same processor, if memory constraints allow. Thereafter the cost values in the cost matrix are updated after each run so that they always represent the cost of the current allocation selected. This process is repeated while the allocation, generated by the allocation algorithm,continues to improve.

Our research has suggested that the cost function consists of varying measurements depending on the type of program being distributed, i.e. if, in a particular program, the level of communication is high over a number of different channels, then the main emphasis should be placed on the number of data units transferred over each channel, while if the opposite is true, i.e. a small amount of communication over only a few channels, then the total process execution time is a good enough measure. Although the current implementation does not require this classification, it was seen as a possible extension and would involve the user classifying his particular application program according to the nature of the inter-process communication, i.e. the granularity of the application must be identified. Granularity refers to the amount of time being spent on communicating versus computing in a parallel program [OBE88]. In a course grained application, the parallel processing system consists of large independent chunks with little time - of the order of hundreds of communications per second between processors - spent on communicating between the individual processors. In a fine-grained application, more time (millions of communications per second) is spent on communicating and synchronizing between the processors.

Communication is thus regarded as the most important cost because it is the most expensive. This is due not only to the fact that the network buses are potential control and traffic bottlenecks, but also that they may be the least reliable part of the system.

Because, as discussed in Section 5.1. above, the matcher process need not be on the same processor as either of the two communicating processes, the cost of two processes communicating may not be as expected. As a result, it is preferable that the possible positions of the matcher process be taken into consideration when developing tne cost function.

Other factors which may be included in the cost function are:
- the proportion of total run-time that a processor is idle.
- load balancing i.e. when one computer sits idle while another is overworked [AND88].
- the proportion of execution time that is spent on communication.
- the number of times a process is given a chance to run but is busy waiting to communicate.

Objective functions other than total running time can be considered. There are many suitable cost functions one may wish to use. For example, instead of absolute time, one may choose to minimize financial costs. For this objective function, the inter-process communication costs are measured in costs incurred per communication, and the running time costs of each process are measured in cost for computation time on each processor, taking into account the relative processor speeds and the relative costs per computation on each processor. Many other useful objective functions can be met by choosing the cost function appropriately.

7. The Integer Programming Method

The *integer programming method* is based on an implicit enumeration algorithm[6]. A set of decision variables is defined and the problem is formulated as an objective function of these variables to be minimized (or maximized), and a set of constraints to be observed for the generated values of these decision variables [ELD80]. Constraints may be easily incorporated into the allocation model, but the algorithm is limited by the amount of time and memory needed to obtain an improved solution. The model minimizes total processing cost subject to such resource limitation constraints as memory capacity and real-time requirements [MAP82] [CHE80].

7.1. Implementation

The design of the mathematical model for the process placement involves three major steps:

1. the formulation of a *cost function* to measure inter-processor communication (IPC) cost and processing cost,
2. the formulation of a set of *constraints*, and
3. the derivation of an iterative *algorithm* to obtain a minimum total cost solution.

7.1.1. Objective Function

The cost function is formulated as the sum of the IPC cost and the processing cost. IPC cost is a function of both process coupling factors and inter-processor distances. If processes i and j are assigned to processors k and l, respectively, $q_{ik} = (c_{ij} * d_{kl})$. Coupling factor C_{ij} is the number of $\text{data}_{\text{units}}$ transferred from process i to process j. Inter-processor distance d_{kl} is certain distance-related communication costs associated with one unit of data transferred from processor k to processor l. The inter-processor cost is $(C_{ij} * d_{kl})$. Processing cost q_{ik} represents the cost to execute process i on processor k.

7.1.2. Constraints

Several constraints are incorporated in the allocation model to achieve the load balance and meet the application requirements. They include bounded attributes, process preference, process exclusivity and process redundancy. Bounded attributes are the set of constraints associated with the application processes and given network topology. For example, the memory attribute is represented by

$$M_i <= S_k$$

where M_i is the amount of memory required by process i, and S_k represents the memory capacity at processor k. This attribute states that the amount of memory required for all processes assigned to a processor must not exceed the processor memory capacity.

6. *Enumeration techniques examine all feasible alternatives.*

The *process preference matrix* indicates that certain processes can only be executed by the specified processor. It is represented by an m*n matrix P, where $P_{ik} = 0$ implies that process i cannot be assigned to processor k; and $P_{ik} = 1$, otherwise.

The *process exclusive matrix* defines mutually exclusive processes. It is represented by an m*m matrix E, where $E_{ij} = 1$ implies that processes i and j cannot be assigned to the same processor; otherwise, $E_{ij} = 0$.

Process redundancy may be provided for system reliability. This permits multiple copies of a process. For example, if process i has a redundancy of three, new processes i+1 and i+2 are added to the original set of processes.

7.1.3. Algorithm

The algorithm is derived from a distributed enumeration technique, the branch and bound (BB) method. This method consists of a set of rules which constitute a depth-first search. The rules are used to select and expand the next node, to eliminate a node, i.e. to prune the tree, and to terminate the algorithm. (The algorithm is terminated when all possible paths have been investigated or the limit for the number of iterations has been reached. This limit exists to enable trading a much improved solution for a fast algorithmic result.) To employ the BB technique, the placement problem is represented by a search tree. The placement decision represents a branching at the node corresponding to the given process.

Consider the problem of allocating m processes among n processors. Starting with process 1, each process is allocated to one of the n processors subject to the constraints imposed on the relations on processes and processors. (The number of tree levels m corresponds to m processes.) A feasible sequence of successive branches is called a path. A path from the root node to the last node represents to a complete allocation; otherwise, it is a partial allocation. The cost of a path is computed according to the cost function. An example of a three-level search tree with three processors is shown in Figure 9.

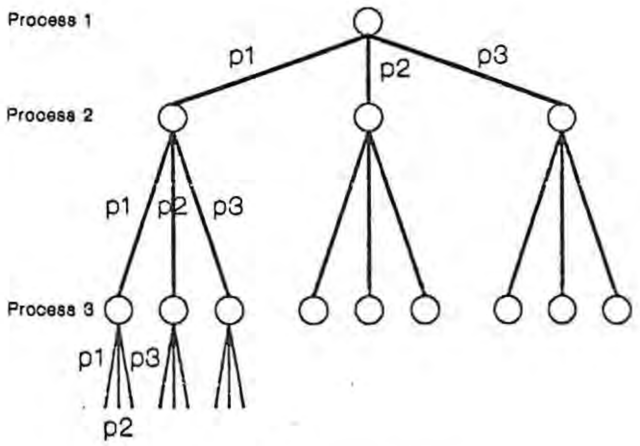
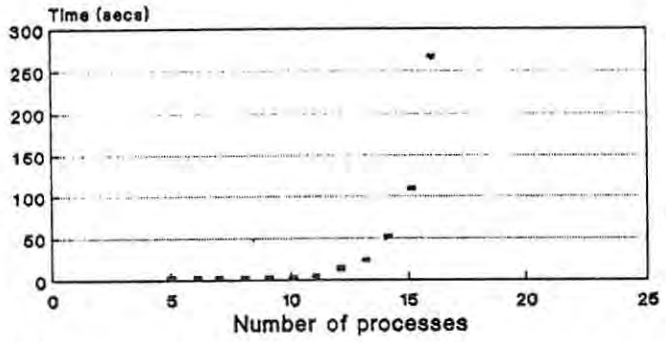


Figure 9 - a search tree

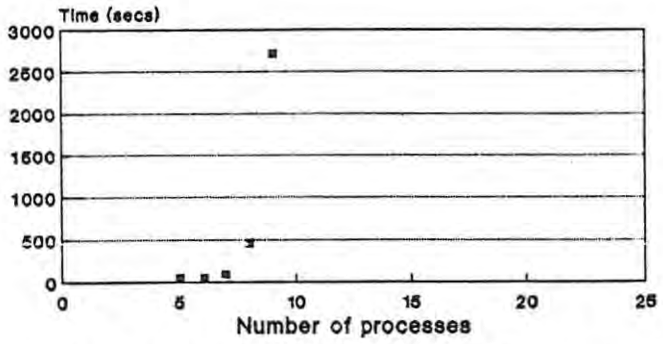
Using a BB method to find a solution in the search tree is known to be an NP-complete problem [KAS84]. However, imposing a large number of constraints on the BB technique greatly reduces the feasible solution space. This technique is best suited to a loosley-coupled distributed system whose IPC cost is high. The model is able to accommodate a large number of constraints and thus a good load balance can be achieved by the suitable setting of input parameters such as the preference matrix and exclusive matrix. The allocation model was designed to be a general purpose model. When a specific requirement exists in the application, the inputs to the model must be tailored. To identify all application requirements is thus an integral part of the successful use of such a model.

7.2. Run-Time Performance

The branch and bound method[7], generates, for a given distributed system configuration and set of constraints, a solution for medium-sized problems satisfying all application requirements. The model generates a very effective allocation although the execution speed of the algorithm is such that it is not viable for real-time systems. This is clearly shown in Figure 10a and b where the algorithm was required to allocate a varying number of processes over a 2 and 5 processor system, respectively. As the number of processes is increased, the performance in execution speed becomes incrementally worse.



(a) the result of distributing a suite of test programs with varying no. of processes on a 2-processor network



(b) the result of distributing a suite of test programs with varying no. of processes on a 5-processor network

Figure 10 - the run-time performance of the Integer Programming algorithm

7. *This Integer Programming method was written in Turbo Pascal (Ver. 5.0) and implemented on an IBM compatible AT, running at 8MHz.*

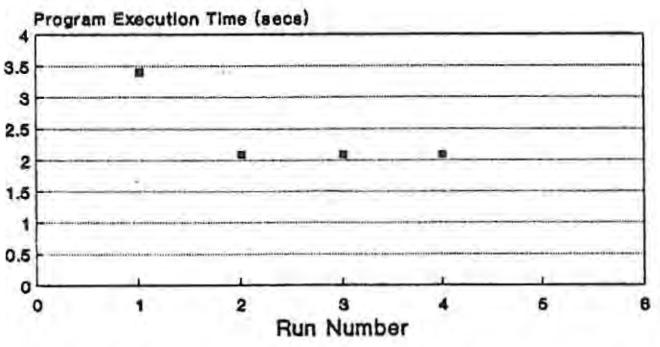


Figure 11 - the average improvement in execution speed of a suite of test programs after repeated use of the Integer Programming algorithm

Figure 11 shows the average improvement in execution speed of a suite of distributed test programs[8] where the process allocations were varied. It is important to note that only a few iterations of the placement algorithm are necessary before a marked improvement in the execution speed of the test program is reached. The test programs were contrived to be relatively busy by forcing processes to repeatedly perform expensive mathematical operations and inter-process communication.

8. The Graph Theoretical Approach

The graph theoretical approach represents the processes to be allocated as a set of nodes in a graph. The inter-process communication (IPC) cost is represented by the weight of a non-directed arc connecting the two nodes. An inter-process cost of zero means no communication takes place between the two processes and therefore they are not connected in the graph. An IPC cost of infinity means these two nodes must be assigned to the same processor. The arc weights then represent the cost of communication between a pair of modules not co-resident on a processor. Any pair of co-resident modules is assumed to have zero IPC cost. The algorithm minimizes the total IPC cost and total processing cost by performing a max-flow/min-cut algorithm[9] [TAN81] on the graph.

8.1. Implementation

The process placement strategy in this model is to minimize total cost, defined as the sum of processing cost and IPC cost.

8. *All the test programs written to compare the effectiveness of the three process placement algorithms implemented for this project, were written in an amended version of occam as designed by [HIL88].*

9. *A max-flow/min-cut theorem states that, in any network, the value of any maximal flow is equal to the capacity of any minimal cut.*

Processing cost is given by the Q matrix

$Q = \{q_{ik}\}, i=1,...m, k=1,...n$

where q_{ik} represents the processing cost for process M_i on processor P_k. We assume this cost is available. It is a measure of the processing requirements of the process. A value of infinity q_{ik} = &, implies that process M_i cannot be executed on processor P_k.

Let c_{ij} = IPC between processes M_i and M_j. The total cost for processing a given process can then be expressed as a function of the allocation, X,

$$\mathrm{COST}(X) = \sum\{q_{ik}x_{ik} + \sum c_{ij}x_{ik}x_{jl}\}$$

where ,

$q_{ik}x_{ik}$	=	processing cost for each process on its assigned processor , and
$\sum c_{ij}x_{ik}x_{jl}$	=	IPC cost between non-critical processes

Each possible placement corresponds to a cutset of a graph, such that if the cutset partitions a node into the subset containing the node representing processor 1, then the corresponding process is assigned to processor 1. The optimum placement corresponds to a minimum weight cutset. The minimum cost allocation is obtained by performing a min-cut algorithm on the graph.

A maximum flow algorithm is used to find a minimum cost assignment.

To illustrate, consider the example from Stone [STO77] shown in Figure 12. The process consists of 6 modules, A,B,C,D,E,F. The graph is constructed with nodes for each of the processes, and IPC costs on the arcs joining the nodes. In order to represent processing costs, two additional nodes are added to represent the two available processors, p1 and p2. These two nodes are regarded as the source and sink nodes, respectively, in the resulting graph. The cost of running each process on processor p1 is denoted on the arc joining that process node to node p2. If we perform a min-cut algorithm on the graph we obtain the cut shown by the heavy dark line. This provides the minimum cost allocation of the given processes between two processors.

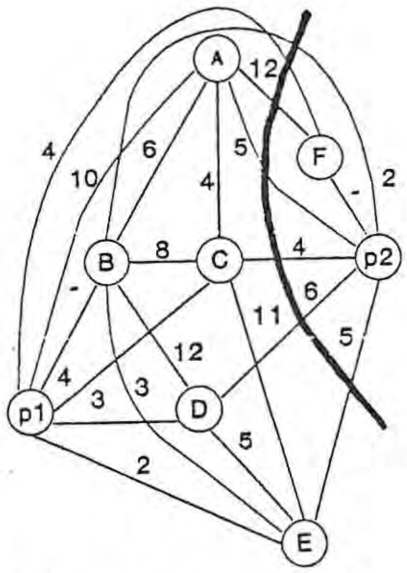


Figure 12 - an example graph

The maximal flow problem may be solved using linear programming techniques, but if the graph consists of too many nodes, then a labeling method, devised by Ford and Fulkerson [COO74] [BEC64], is adopted. A brief outline of this algorithm is given below.

The maximal flow algorithm may be organized into a simple N*N matrix, where N is the number of nodes in the resulting graph. The elements of the matrix are initialised to the values of all the arcs in the graph. Three extra columns are needed in the matrix. The first, labelled ***alpha***, records the excess capacity (the amount of communication recorded on each arc), $d_{\text{source node,current node}}$, the second, labelled *beta*, records the identity of the source node, and the third, labelled *step*, records the current stage at which the labeling process is at.

Let (i j) = the arc connecting node i to node j, and

d_{ij} = excess capacity on arc (i j).

Beginning in row 1 (the source), find all columns with $d_{ij} > 0$. For each such column, $\text{alpha}_j = d_{ij}$ and $\text{beta}_j = 1$. As long as the Nth row (sink) has not been labelled, repeat the following. Begin at the first row of the set of rows just labelled. Find all columns with $d_{ij} > 0$, for which row j has not been labelled. Row j is then labelled with $\text{alpha}_j = \min(d_{ij}, \text{alpha}_i)$, and $\text{beta}_j = i$.

If the entire matrix has been scanned and the sink has been labelled then it means that the maximal flow can still be increased further. To do this the entire matrix must be updated and the labeling procedure repeated.

A simple example has been selected to demonstrate this procedure, although the run-time performance of the algorithm is measured using more complex systems. The example consists of three processes which need to be allocated to two available processors. The costs of each process executing on each of the processors is known and is given below. The inter-process communication costs are filled in on the resulting graph which is shown in Figure 13. The costs of execution are as follows:

	Processor S1	Processor S2
Process A	10	1
Proces's B	1	10
Process C	10	1

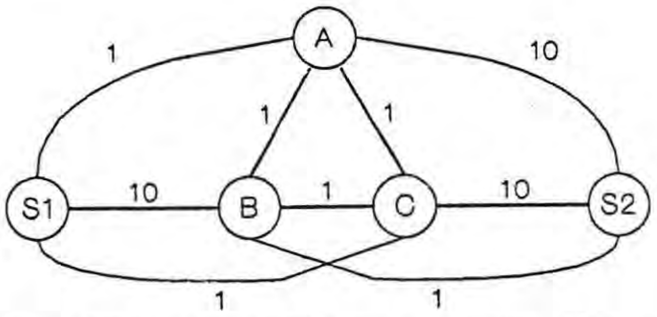


Figure 13 - graphical representation of communication and execution costs

	S1	A	B	C	S2			
S1	0	1	10	1	0	99	99	0
A	0	0	1	1	10	1	1	1
B	0	0	0	1	1	10	1	1
C	0	0	0	0	10	1	1	1
S2	0	0	0	0	0	1	2	2

(a)

	S1	A	B	C	S2			
S1	0	0	10	1	0	99	99	0
A	1	0	1	1	9	0	0	0
B	0	0	0	1	1	10	1	1
C	0	0	0	0	10	1	1	1
S2	0	1	0	0	0	1	3	2

(b)

	S1	A	B	C	S2			
S1	0	0	9	1	0	99	99	0
A	1	0	1	1	9	0	0	0
B	1	0	0	1	0	9	1	1
C	0	0	0	0	10	1	1	1
S2	0	1	1	0	0	1	4	2

(c)

	S1	A	B	C	S2			
S1	0	0	9	0	0	99	99	0
A	1	0	1	1	9	0	0	0
B	1	0	0	1	0	9	1	1
C	1	0	0	0	9	0	0	0
S2	0	1	1	1	0	0	0	0

(d)

Figure 14 - the stages in the labeling procedure for the graph in Figure 13[10]

To update the matrix,

if $\text{beta}_N = r$

then row N was reached from row r

$$\begin{aligned} d'_{rN} &= d_{rN} - \alpha_N \\ d'_{Nr} &= d_{Nr} + \alpha_N \end{aligned}$$

For example, in the labeled matrix from Figure 14a,

$$\begin{aligned} d'_{2,5} &= d_{2,5} - 1; & d'_{5,2} &= d_{5,2} + 1; & d'_{1,2} &= d_{1,2} - 1; & d'_{2,1} &= d_{2,1} + 1 \\ &= 10 - 1 & &= 0 + 1 & &= 1 - 1 & &= 0 + 1 \\ &= 9 & &= 1 & &= 0 & &= 1 \end{aligned}$$

Further, if $\text{beta}_r = s$

then row r was reached from row s

$$\begin{aligned} d'_{sr} &= d_{sr} - \alpha_N \\ d'_{rs} &= d_{rs} + \alpha_N \end{aligned}$$

This updating procedure is repeated until $\text{beta}_x = 0$.

If the stage is reached where no more rows can be labelled and the sink has not been labeled, then the maximal flow has been reached. The maximal flow is calculated as the sum of all elements in the matrix in column 1 or

10. *The stages depicted here are identical to those calculated and output by the implemented algorithm during the course of its attempt to arrive at a solution.*

row N. Once again, referring to the example above, in the last matrix, Figure 14d, the sum of the elements in the last row or first column is 3.

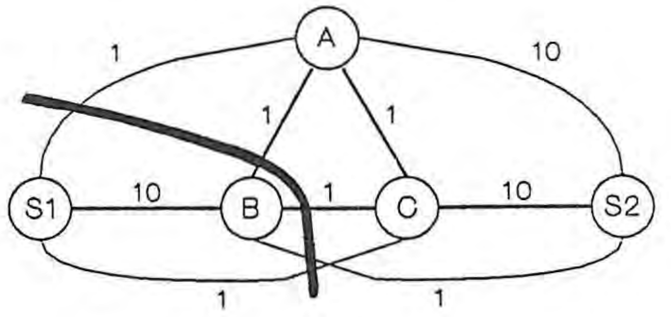


Figure 15 - a complete graph

The actual placement of processes is obtained by noting in the final table those rows which are labelled and those that are not. Those rows that are labelled represent processes that must be assigned to the one processor, and those rows that are unlabeled represent processes that must be assigned to the other processor. This rule only applies to a two processor system.

8.2. Extension of the Algorithm to more than Two Processors

The labeling algorithm may be generalised to deal with an n-processor system by extending the network flow approach. We begin with the extension to a 3-processor system.

Firstly, the notion of a cut set must be extended. Stone and Bokhari [STO77] define a *tricutset* in a graph with three source/sink nodes as *a subset of arcs whose removal partitions the graph into three distinct subgraphs, such that each source/sink node lies in a distinct subgraph*. A tricutset for the network of Figure 16 appears in Figure 17.

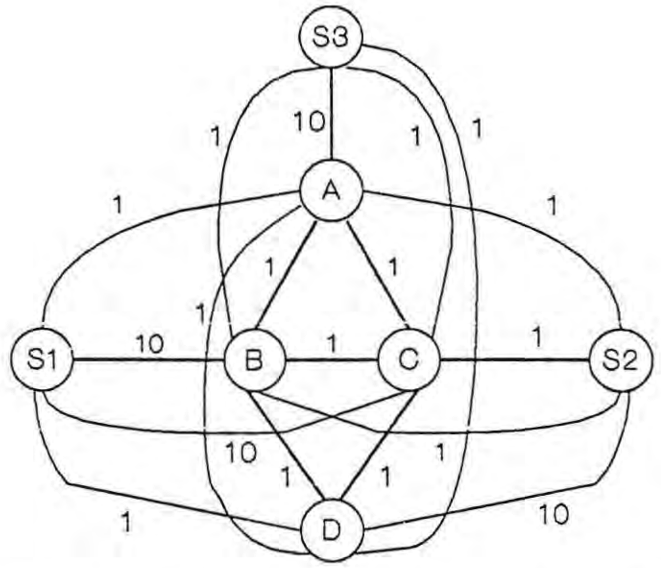


Figure 16 - Inter-process and process-processor connections

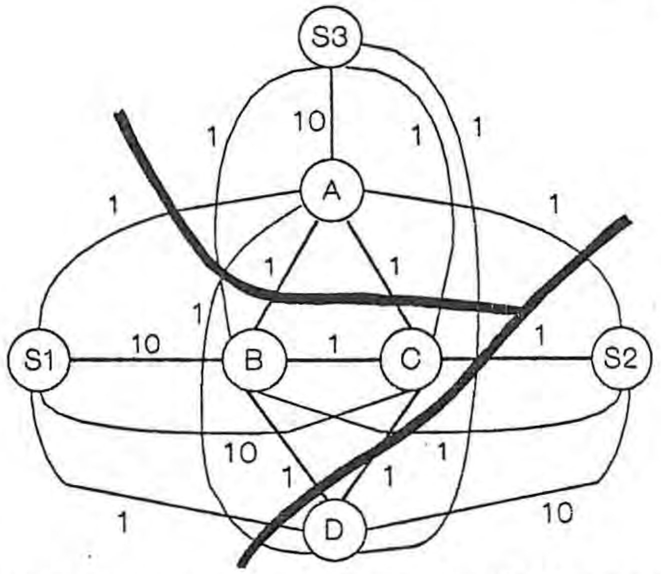
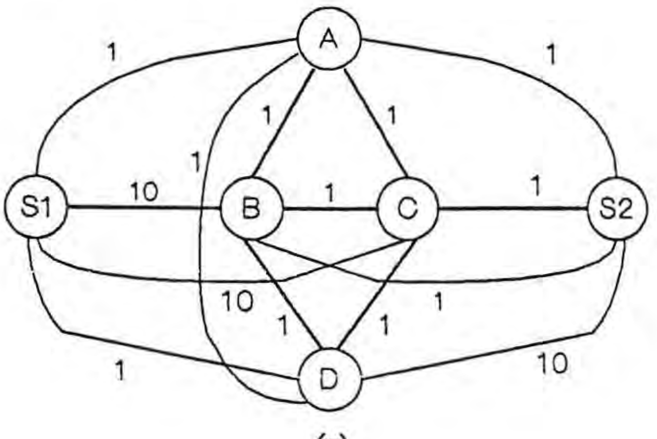


Figure 17 - a tricutset representing a process placement with a three processor network

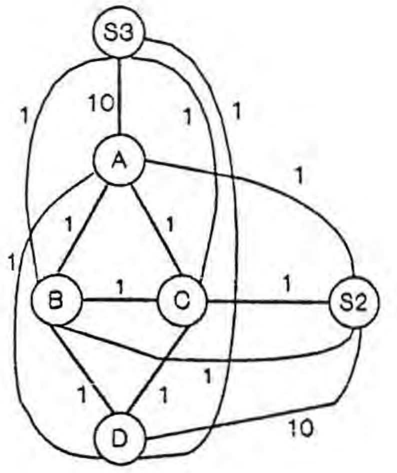
The major difference in the n-processor system is the calculation of the cost that results from a process running on a particular processor. Suppose that process D runs in time T_i on processor p_i, i = 1,2,3. Then the edge from node D to node S_1 carries the weight $(T_2 + T_3 - T_1)/2$, and likewise the edges to nodes S_2 and S_3 carry the weights $(T_1 + T_3 - T_2)/2$, and $(T_1 + T_2 - T_3)/2$, respectively. This idea generalizes naturally to n-processors. The running time of D on P_2 contributes to the weight of the branches from D to every node S_i, i = 1,2,...,n. Its contribution to the branch to S_1 is $-(n-2)T_1/(n-1)$. Its contribution to the branch to S_i, i< >1, is $T_1/(n-1)$. In this way we still obtain the desired property that the weight of a cutset is equal to the cost of the corresponding module placement.

The basic idea behind the algorithm for finding the minimum tricutset in a three-processor graph, is to run a network flow algorithm, in our case the Ford and Fulkerson labeling algorithm, between nodes S_1 and S_2, then between S_1 and S_3 and finally between S_2 and S_3. The three subgraphs resulting from the graph in Figure 16, are given below in Figures 18a,b and c.

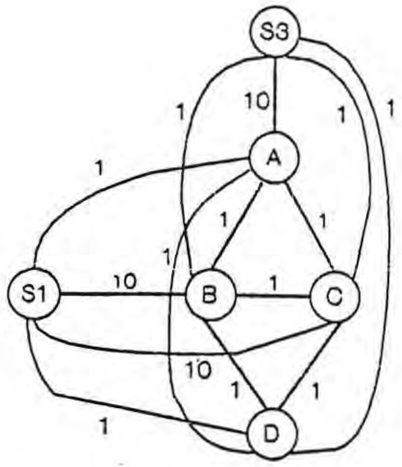


(a)

Figure 18 - subgraphs for n > 2 processor system



(b)



(c)

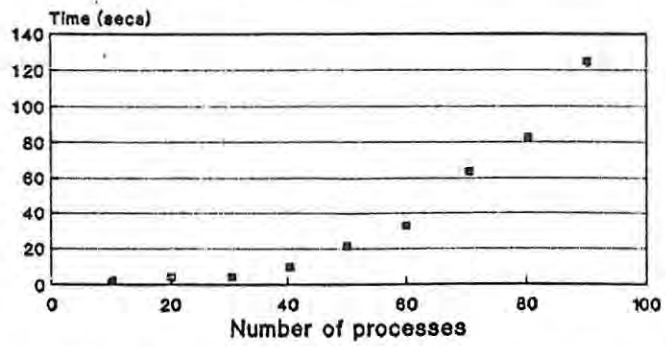
Figure 18 - continued

The extension of this method to an arbitrary number of processors requires an n-dimensional min-cut algorithm which quickly becomes computationally intractable. This limits the usefulness of the method in many applications. However, an algorithm to deal with four or more processors in the system has been proposed for cases where the pattern of inter-process communication costs can be constrained to a tree. This algorithm is discussed in Section 11.3.

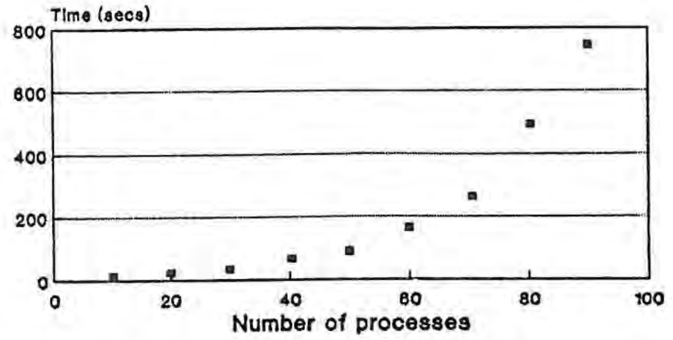
Nonetheless, some problems with the n-processor placement do remain unsolved. Among these are the fact that a node might not be associated with the same processor in a minimum n-processor cutset as it is in a two-processor cutset.

8.3. Run-Time Performance

The same suite of test programs as were used to test the Integer Programming approach, exhibiting the same characteristics, were used to test the efficiency of the Graph Theoretic approach.



(a) the result of distributing a suite of test programs with varying no. of processes on a 2-processor network

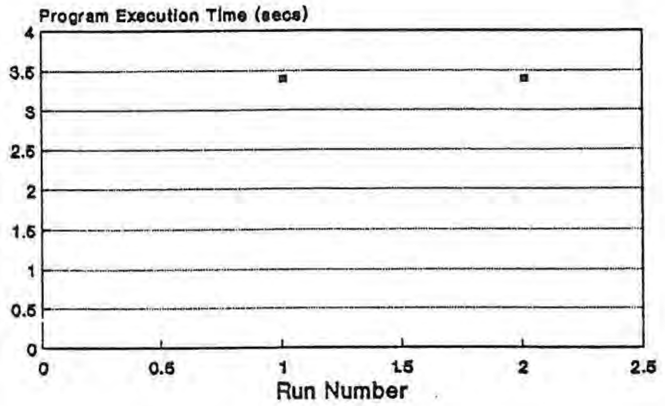


(b) the result of distributing a suite of test programs with varying no. of processes on a 5-processor network

Figure 19 - the run-time performance of the Graph Theoretic algorithm

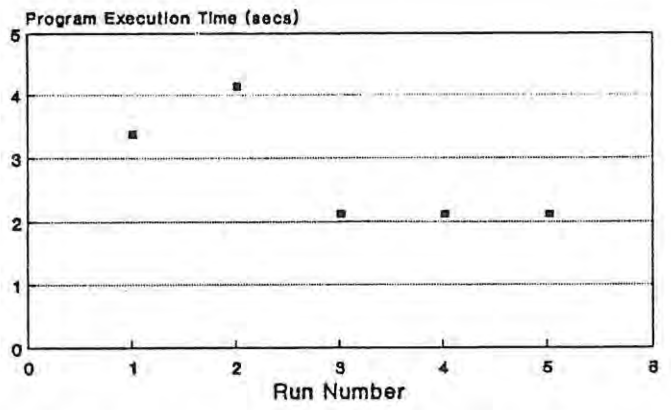
As Figures 19a and b exhibit, this approach finds the minimum cutset very efficiently for almost all graphs. There are some graphs, however, for which the algorithm[11] fails to find a good solution. A problem arises when a network of more than two processors is used. The algorithm becomes rather labourious and time consuming, and if the network is not homogeneous, then the execution time of each allocated process must be obtained. These times are usually estimated, and this explains the increase in execution time of the program before an improvement in the allocation is achieved. An example of this pattern is shown in the graph of Figure 20b.

Difficulty in incorporating various constraints prohibits the generalization of this model. Consequently, there is no mechanism for load balancing or for including limited resources such as memory size or processing time. This often results in suboptimal allocations being generated. These short-comings suggest that the graph theoretic approach is not the best solution to the process placement problem. Figure 20a is an example of where a suboptimal allocation, which is the same as the initial allocation, was generated. The program exhibited a coarse granularity.



(a) the result of distributing a suite of coarse-grained programs over a homogeneous processor network

11. *The Graph Theoretic approach was written in Turbo Pascal (Ver. 5.0) and implemented on an IBM compatible AT, running at 8MHz.*



(b) the result of distributing a suite of finely-grained programs over a non-homogeneous processor network

Figure 20 - the average improvement in execution speed of a suite of test programs after repeated use of the Graph Theoretic algorithm

9. The Heuristic Method

The basic idea of heuristic methods is to find an allocation that will balance IPC costs and load balancing by including a measure of association among interacting processes. This technique requires much less computation time than integer programming methods. As a result, it can solve time-critical cases and also larger dimensional problems [CHU80] [GYL76].

The formulation of a performance measure based on the total amount of interaction among computational processes requires the definition of a quantitative measure of such interactions. To construct a performance measure, we proceed as follows.

x_{jr} = assignment variable
= 1 if the jth process is assigned to the rth processor; 0 otherwise
N_E = number of processors in the system
N_P = total number of processors under consideration
M_r = memory size of the rth processor
T_r = $T \times a_r$
= length of a time-slice allocated to the rth processor
s_j = amount of memory storage required by the jth process
t_j = amount of processor time required by the jth process
c_{ij} = measure of the amount of inter-process communication between ith and jth processes
= message time
= number of messages exchanged

$$
\begin{aligned}
Q &= \sum_{j=1}^{N_P}\sum_{r=1}^{N_E}\sum_{i=1}^{j} c_{ij}x_{ir}x_{jr} \\
&= \text{total amount of bus traffic eliminated (or intra-processor communication generated)} \\
&\quad \text{by making processes share the same processors}
\end{aligned}
$$

Optimal Workload Partitioning Problem:

Maximize Q subject to the following constraints:

$$
\begin{aligned}
\sum^{N_P} s_j x_{jr} &\leq M_r \text{ for } r = 1,\ldots,N_E \text{ (processor memory constraint)} \\
\sum^{N_P} t_j x_{jr} &\leq T_r \text{ for } r = 1,\ldots,N_E \text{ (processor time constraint)} \\
x_{jr} &= 0 \text{ or } 1
\end{aligned}
$$

Cluster analysis is an example of a heuristic method whereby the workload partitioning is seen as essentially a classification problem in which the objects to be classified are the processes that inter-communicate. The aim is to group these objects into clusters such that the level of communication among clusters is minimized.

To allow for the use of the cluster analysis technique, the model of the optimization problem is expanded to include the measure of association among interacting processes. Selection of a proper measure, and of suitable measurement variables, is important.

9.1. Implementation

In a heuristic approach to process placement, some simplifying assumptions about processes are made. Given a set of processes, and the cost of data transfers among them, (subject to constraints such as limited processor memory size and real-time considerations) heuristic models form process clusters with a minimum of inter-cluster communications.

9.1.1. Approach 1

A simple classification algorithm is presented by Gylys et al [GYL76] that searches for pairs of modules such that when these modules are assigned to the same processor, the greatest inter-processor communication cost is eliminated. This "fusion" process continues until all possible pairs are fused.

The algorithm assumes that the number of processors is known, and the objective is to form module clusters with a minimum of inter-cluster communication. When each of these clusters is assigned to a processor, inter-processor communication is minimized. To form such clusters, a module pair with the maximum inter-process communication cost is found, and is then checked to see whether or not a single processor can handle it. If the constraints are satisfied, the pair is fused into a single process, otherwise the pair with the next highest inter-process communication cost becomes a candidate cluster. The process ends when all possible pairs are fused.

The algorithm is tabulated below:

1. Set $N_E = N_P$.
   Assign one process per processor.
   Mark every pair of processors as being eligible for "fusion" into a single processor.
   Construct a list of all eligible pairs.
2. Find an eligible pair of processors whose 'fusion' into a single processor would eliminate the greatest amount of bus traffic.
3. Can the combined load already assigned to the candidate pair of eligible processors be handled by a single processor? If 'Yes' go to 4; else go to 5.
4. Fuse the processors of the selected pair into a single processor and combine their workloads. Go to 2.
5. Eliminate the pair.
6. Are there any eligible pairs left? If *Yes* then go to 2; else stop.

An important limitation of this algorithm is that there is no mechanism to guarantee that the number of clusters found will not be more than the number of available processors. If this is the case, then the excess clusters must be divided among the available processors.

9.1.2. Approach 2

This approach is also suggested by Gylys et al [GYL76]. The algorithm defines a distance function between processes. If process M_k communicates with process M_l, then the two processes exhibit a degree of similarity which may be measured by the volume of data v_{kl} that is transferred between them.

In the algorithm, 'initial centroids'[12] are first assigned to each candidate cluster. The distance from each process to the centroid of each cluster is calculated. A search is then made for any pair, consisting of a process and a cluster centroid, with the smallest distance between them. Assigning the process within the chosen pair to the cluster from the chosen pair will make for the greatest reduction in the total inter-process communication cost. When a pair is selected it is checked to see whether it satisfies the constraints. If so, the placement is made. Otherwise, the pair with the next smallest distance becomes a candidate. Whenever an placement is made the cluster centroid is adjusted. It is possible for a process, which has already been assigned to a cluster, to be reassigned once the centroids have been adjusted. The process continues until no process clusters change, or until the number of iterations exceeds a preset limit.

12. *A centroid is a mean value - in this case a mean value of communication volume.*

A method is given below of how a space of quantitatively measurable associations can be defined for interacting processes.

$$[M] = [N_P]\,[N_M]$$

where ,

$[N_M]$	=	total number of distinct message types used for inter-process communication, and
$[N_P]$	=	number of processes in the model.

$$d(i,j) = \frac{\sum_{k=1}^{N_M} \mid |m_{ik}| - |m_{jk}| \mid}{\sum_{k=1}^{N_M} \mid |m_{ik}| + |m_{jk}| \mid}$$

where ,

m_{ik}	=	$r_i M_k$ if the ith process produces (consumes) the kth message
	=	0 otherwise
$r_i M_k$	=	measure of the contribution of the kth message to bus loading per time-slice due to process i
M_k	=	measure of the message length
r_i	=	production/consumption rate of the kth message by the ith process.

Intuitively, the distance function d measures the degree of similarity (or intensity of interaction) between two processes. The more similar two processes are, the smaller the value of d.

A non-hierarchical clustering method was selected, which is based on MacQueen's k-means algorithm [TRY70] [EVE74] [VAN77] [DUR74] [AND73].

The clustering algorithm used consists of the following steps:

1. assign an initial centroid for each candidate cluster, ie. processor,
2. for each process to be assigned, compute the distance from it to the centroid of each cluster,
3. assign the process to the nearest cluster and subsequently adjust the centroid, and
4. repeat steps 2 and 3 until no processes remain to be allocated.

A disadvantage of this approach as well as of Approach 1, is that the search is carried out for a long time before an eligible pair of processes and/or cluster is found. The length of the search is a result of the need to find the pair with maximum communication cost or minimum distance. To do this requires an examination of the communication costs between every process pair. Also, the real-time constraint does not consider the queueing delays or precedence relationships, factors which have a great influence on the process placement problem as discussed in Chapter 11 below.

9.1.3. Approach 3

Efe [EFE82] suggests yet another algorithm which he feels eliminates the shortcomings of the abovementioned approaches.

First, a process clustering algorithm obtains the minimum inter-processor communication cost without constraint considerations. The result of this algorithm may also satisfy the load-balancing constraint. If not, the overloaded and underloaded processors are identified. Some processes are then shifted from overloaded to underloaded processors by a process reassignment algorithm. If too many processes are shifted by this algorithm, resulting in an overloading of a previously underloaded processor, the algorithm will repeat until the solution arrives at a balanced load distribution.

The *process clustering algorithm* involves a search for a process pair having the greatest volume of data transfer between them. When a process pair satisfying the above requirement is found, it is fused into a single process. This procedure continues until a fusion is made and until the number of processes is reduced to a value which is less than or equal to the number of available processors.

It is suggested that to aid the *process reassignment algorithm*, a graph be developed of the processes of a given application program, how they inter-communicate, and how they have been clustered as a result of the process clustering algorithm. (The graph is similar to those developed in the graph theoretic approach discussed above.) The algorithm proceeds with the identification of all overloaded and underloaded processors [WAH84]. From the graph, all processes which are assigned to processors that have acceptable loads are deleted. If the load on a processor is too great, then the processes assigned to it are retained. If the load on a processor is too small then the processes assigned to it are grouped into a single process.

These two algorithms then form the backbone of the overall algorithm. The process placement scheme is divided into two phases:

Phase 1.

1. For I := 2 to n, where n = no. of processors,
   a) form I process clusters,
   b) assign these clusters to separate processors so that load balancing is maximized.
2. Select an assignment made in 1b) above which maximizes load balancing among those assignments made.
3. If this assignment satisfies the load-balancing constraint, then stop, otherwise go to step 4.

Phase 2.

4. Identify the overloaded and underloaded processors.
5. Use the process reassignment algorithm to assign some processes from overloaded processors to underloaded processors. Then go to step 3.

9.2. Run-Time Performance

The performance of the Heuristic method was measured using a set of test programs similar to those used in the previous two process placement algorithms.

The algorithm[13] that was implemented, for which run-time results appear in Figure 21, makes use of certain characteristics of the process clustering algorithms described in Approaches 1 and 3. Processes are clustered according to their *distance* values which depend mainly on the volume of communication of each process.

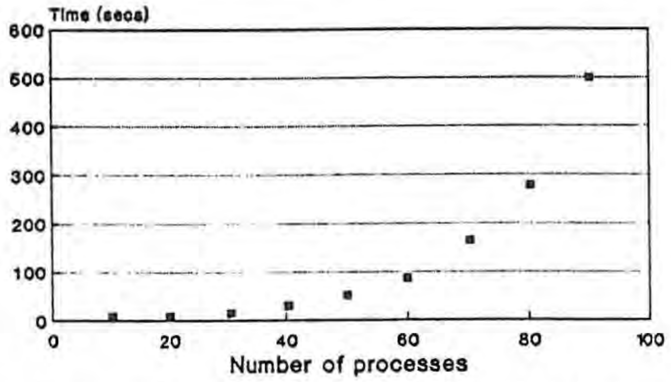


Figure 21 - the run-time performance of the Heuristic algorithm measured by distributing a suite of test programs with varying number of processes on a 2-processor network

The algorithm is given no constraint on the number of processors which must be used in a particular allocation, besides the maximum number of processors available in the whole network. The algorithm suggests an optimal number of processors needed to support the process placement decided upon by the algorithm. Although there are few constraints to be considered by the algorithm in arriving at an allocation, its execution time is slower than expected, although very much improved from the Integer Programming method.

The *heuristic method* provides relatively fast and effective allocations for a suboptimal solution. The suboptimal solution is due to the limited amount of constraints which may be included in this implementation of the algorithm. The literature suggests that the problem does not hold for all implementations of such heuristic algorithms [EFE82] [CHU87c].

13. *The Heuristic method was written in Turbo Pascal (Ver. 5.0) and implemented on an IBM compatible AT, running at 8MHz.*

As can be seen in Figure 22 a good allocation may be reached for simple distributed systems, in this case a 3 process, 2 processor system.

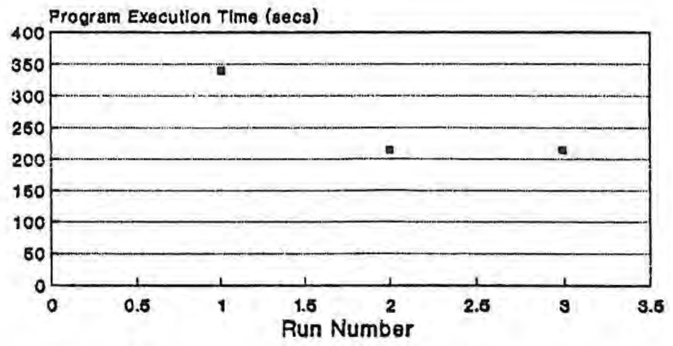


Figure 22 - the average improvement in execution speed of a suite of test programs after repeated use of the Heuristic algorithm

10. Comparisons of Allocation Techniques

Three static process placement algorithms, each with their varying characteristics, have been implemented and tested. Turbo Pascal (Ver. 5.0), running on an IBM compatible, 8 MHz AT, was used as the implementation language.

10.1. The Comparison of Static Allocation Methods

The *integer programming method*, although extremely slow, allows for the easy inclusion and extension of a comprehensive set of constraints necessary for a successful allocation algorithm, e.g. limited processor memory and process preference relationships. As can be seen from Figure 24, the best allocation for a particular class of program and input data, is reached in the least number of runs using this technique. The main disadvantage of this implicit partial enumeration method is its level of complexity with respect to time, as the number of processors in the network, and the number of processes to be allocated, increases.

The algorithm for the *graph theoretic* method is certainly the fastest, and makes efficient use of processing facilities, although a few more iterations of the algorithm are often needed before a good allocation is reached. Factors such as load balancing are impossible to include in the implementation of the algorithm - this accounts for the impressive execution speeds. It would seem that the disadvantage of this method arises when the network on which processes must be allocated, is not homogeneous. In this case, all the processes must be given a chance to run on each processor to gather its execution time statistics before a solution is reached - this obviously becomes too time consuming as the number of processors is increased. Stone [STO77] suggests a means of estimating the execution time of a process on all processors. This is discussed in Section 7 of this report and may be seen as a possible means of overcoming this problem. Once these statistics have been gathered, a much improved allocation is immediately obtained. If the network is homogeneous, then the cost of execution of each process on each processor will be the same, and allocation will be entirely dependent on communication characteristics of the processes. (If processes being allocated execute homogeneous code, then the hardware\software links of occam2 permit arbitrary process placement with no effect on execution speed. However, the moment the code becomes non-homogeneous, then it is important where a process is placed and with what other processes it communicates.)

Non-hierarchical cluster analysis techniques produce good although suboptimal workload partitions for cases where only a few constraints affect the allocation. The results obtained for the *heuristic* method are better than those obtained for the integer programming method, but are disappointing when compared to results obtained by other researchers in the field [EFE82] [GYL76].

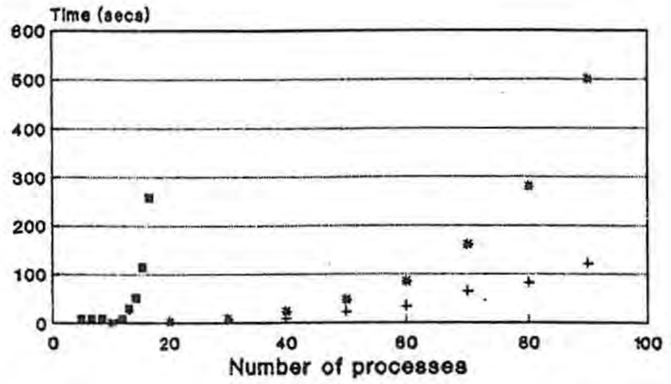
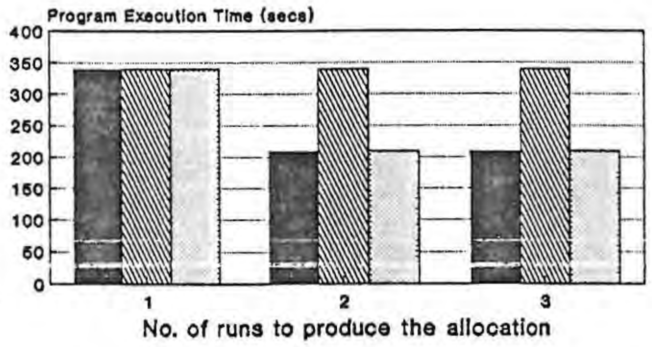
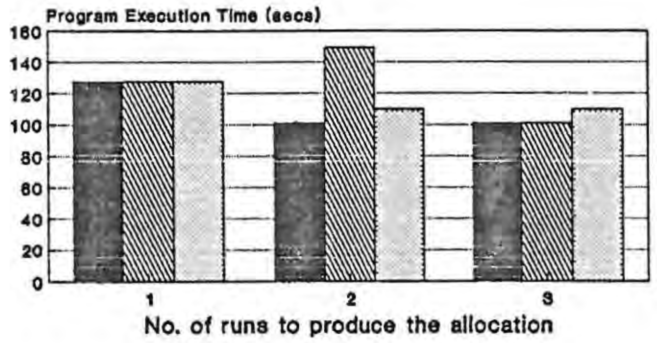


Figure 23 - comparison of execution speeds of each process placement algorithm

Figure 23 shows the marked difference in execution speed of the three process allocation techniques when applied to a suite of test programs with a varying number of processes. Figure 24 highlights the difference in the maximum improvement in execution speed reached by two different suites of test programs which have been repeatedly executed, each time adopting a process placement scheme as suggested by the chosen allocation algorithms. The results shown in Figure 24a were obtained from distributing a suite of test programs, exhibiting a coarse granularity, over a homogeneous network of processors.



(a) a comparison of the results obtained from distributing a suite of coarse-grained test programs on a homogeneous network of processors



(b) a comparison of the results obtained from distributing a suite of fine-grained test programs on a non-homogeneous network of processors

Figure 24 - a comparison of the effectiveness of the three process placement algorithms

Here, the integer programming method was able to generate an improved process allocation after about three iterations of the algorithm, while the graph theoretic approach generated a suboptimal solution almost immediately. A suboptimal solution was in fact the same as the initial allocation, where all processes are allocated to the same processor. The heuristic algorithm generated a good allocation immediately, although this was not the case with many of the more complex programs and data sets. Figure 24b gives results of a more complex distributed system. A suite of test programs, exhibiting a fine grain of parallelism, was distributed over a non-homogeneous network of processors. In this instance, the graph theoretic approach has the problem of having to estimate the running time of each process, of which there were 12, on each of 4 processors. This accounts for the sudden increase in execution time of the second run of the distributed application program before a solution is reached.

It is important to note that the cost of communication on the target system (see Section 5.1.), is high. Therefore, the test programs, written in an amended version of occam [HIL88], were contrived to be relatively busy causing the test results to be slightly artificial.

A trade-off exists between the amount of statistics that must be gathered to enable the algorithm to generate an allocation of processes, and the amount of time taken to gather and process these statistics. Each of the process allocation algorithms discussed uses statistics to a varying degree, and this is evident in the time taken by the respective algorithms. The type of statistics gathered is directly related to the format of the cost function used by the algorithms, and this is another area in which the algorithms differ.

The statistics gathered by the integer programming algorithm include the execution time of each process and the volume of data, measured in units transferred and time taken for each communication. This information, together with factors such as process preferences, and the memory available on each processor, forms the constraints by which the branches of the search tree are developed and pruned. All the information to be gathered and processed contributes to the excessive execution time of this algorithm.

The graph theoretic algorithm does not allow many constraints to be placed on the formulation of a process allocation and as mentioned already, this accounts for the improved execution speed of this algorithm. This does not imply that the algorithm does not generate feasible process allocations, because the statistics gathered on the communication characteristics of the processes are included in the algorithm, and it has already been concluded that communication is one of the most important factors when deciding upon a particular allocation of processes. A consequence is that the allocations suggested by the algorithm are not always the most efficient, and sometimes several iterations of the algorithm are needed before a good allocation is generated.

The heuristic algorithm that has been implemented falls into very much the same category as that of the graph method and the particular approach adopted exhibited similar characteristics. However, heuristic algorithms do exist [EFE82] [CHU87c] that out-perform all other algorithms that have been researched and discussed.

Parallelization of the algorithms which have been studied is a possible means of improving their run-time performance [CHA88] [QUI87], but this area is beyond the scope of this report. Articles by [ELD80], [LAK84] and [SHI81] suggest feasible methods for parallelizing these algorithms with complexities of the order $O(n^2 \log n)$ and $n(n-1 + 2\log_2 n)$, in some cases, reducing execution time by a factor of $O(n)$.

It has been said that fast approximate methods are often indistinguishable from fancy exact 'optimal' methods, especially if the network being used only consists of a few processors and processes, which is the case in the model used for the testing of the algorithms. Furthermore, parallelization and good allocation does not necessarily lead to a marked increase in efficiency.

Whenever an optimal workload partitioning is absolutely required and when the hardware resource constraints are tight, combining two of the methods, i.e. starting the solution process by means of clustering and then continuing it with the implicit partial enumeration method, may constitute a reasonable approach.

10.2. Additional Observations

By varying the number of processors in the network, used for the testing of the process placement algorithms, between 2 and 5, it was noticed that the processing power of the network did not increase proportionally as the number of processors increased (see Figure 25). This fact has been identified and discussed by various researchers in the field of parallel processing [CHU87a] [EFE82] [STE88] [SAN86] and verified by this research.

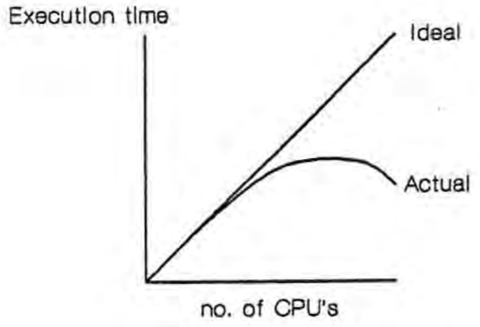


Figure 25 - non-linear speedup in execution as the number of processors is increased

Chu et al [CHU87a] regard program partitioning and process allocation as two major steps in the design of distributed data processing systems, and state that if these two steps are not done properly, an increase in the number of processors in a system may actually result in a decrease in total throughput.

The reasons for the less than linear performance improvement include contention for resources such as memory and their associated buses, and inter-processor communication. This is the so-called *von Neumann* bottleneck [STE88].

The Elxsi 6400 [SAN86] is a message-based machine that achieves linear performance improvement as CPU's are added. This is mainly attributable to the increased efficiency of the operating system kernel as the configuration gets bigger, and is also due to the existence of a cache on each processor to aid the message based communication.

It is also interesting to note how great an effect the power of the processor chip has on the execution speed of a distributed application program. For example, a program consisting of three processes, distributed over a non-homogeneous network of three processors, consisting of an IBM AT, XT, and PC shows a marked difference in execution speed depending on where the processes are allocated (according to a good allocation). This highlights the fact that the processing power of the processors available in the network being used, is an important component of any allocation algorithm.

In the test program run on the non-homogeneous, 3 processor system, two of the three processes had a high volume of inter-process communication, while the third concentrated on matrix multiplication. However, because of the difference in processing power available in the network, the best solution generated by all three algorithms implemented was to place the three processes onto the AT for execution, although it might have been expected that the two communicating processes would be placed on one processor, and the third on another processor. A bar graph of the difference in execution speed of the 3-process program on each of the AT, XT and PC is given in Figure 26. This introduces an interesting fact that parallel speedup is not necessarily a measure of increased performance - it might be the result of differing processor architectures.

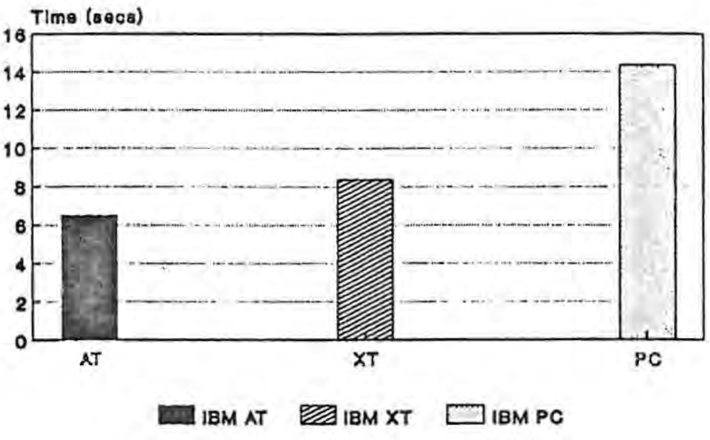


Figure 26 - comparison of processing power

11. Related Topics

As research into the area of process placement continues, so more factors are being discovered which play an important role in the selection of the most efficient process allocation. Some of these factors are discussed below, although their associated implementation has not formed part of this thesis.

11.1. Process Allocation and Precedence Relations

Precedence relationships among processes play an important role in process response time[14], and thus allocation algorithms that take precedence relationships into consideration when deciding upon a particular allocation, generate better process placements than those algorithms that do not. Also, the program-size ratio between two consecutive processes has an effect on whether the two processes should be coresident.

The objective functions and costs considered so far by the algorithms already discussed, are acceptable for non-real-time applications, but for real-time systems, response time is the most important performance measure [CHU87b] [CHU87d]. Minimizing inter-process communication alone, may not produce a good placement.

The processor load consists of loads due to process execution and inter-process communication. Therefore, both execution time and inter-process communication play important roles in process placement and influence response time.

It is proposed to use the workload of the bottleneck processor as the objective function for process placement, i.e. the placement is required that yields the minimum bottleneck among all possible placements.

Chu et al have conducted several experiments to study the effect of precedence relationships on process allocation. The results of these experiments were subsequently used in the development of a process allocation

14. *The reasons for this are explained and supported by examples in Chu et al [CHU87b].*

algorithm. Two principles were also deduced from these experiments. The first states that assigning two consecutive processes to the same processor yields good response times if the execution time of the second process is much larger than that of the first process. Conversely, the second principle states that if the second process is much smaller than the first one, separating the two consecutive processes and assigning them to two different processors, yields a better response time.

A *heuristic algorithm* is developed that considers precedence relationships, inter-process communication and process execution times to search for the minimum bottleneck placement. The precedence relationship among processes specifies the execution sequence of processes.

The algorithm consists of two phases, similar to those suggested in Approach 3 (Section 9.1.3.) above, except that precedence relationships are also taken into consideration when deciding upon the clustering of the processes.

The effect of precedence relationships on response time may be in conflict with the effect of inter-process communication. Therefore, the allocation algorithm jointly considers the effects of both.

Experiments have proved that the allocations produced by this technique are as good, and in some cases better, than those allocations generated by an exhaustive search.

11.2. Process Placement and Process Replication

A new algorithm has been developed, by Chu et al [CHU87c], to iteratively search for process placements and replications that reduce process response time. The algorithm takes the standard costs of communication and execution time into consideration, but in addition, process precedence relationships introduced in the previous section also play an important role in the allocation process. An important factor which is also introduced, is that of minimizing thread response time requirements (a thread is a sequence of processes).

A further shortcoming of previous approaches to the placement problem is that multiple invocations of a process are not taken into consideration by the placement algorithm. As a result, the queueing delay from multiple invocations, which is a significant portion of response time, is ignored. A means of approaching this problem is to selectively replicate processes on processors according to loading constraints. Each invocation for a replicated process is then routed to an appropriate processor. A special algorithm is required to perform routing and may include some form of 'round-robining'.

The process placement proposed considers repeated invocations, queueing effects, process precedence relationships and process replications. The replicated process placement problem minimizes program response time by:

1. determining the optimal number of copies of each process, and
2. allocating these process copies to processors such that system performance objectives are satisfied.

The usual factors of the number of processes versus the number of processors in the system, are parameters to the placement algorithm, but network delays, process invocation rates, and the control-flow graph of the application program are also considered.

Common methods of tackling such an optimization problem, some of which have been discussed in detail in previous sections, are approximation algorithms, probabilistic algorithms, branch and bound and local search techniques. Due to the complexity of the proposed algorithm, a search is made for local solutions and the final solution is selected from this set of local optimums.

The algorithm consists of three major components. Each is briefly outlined below, and the reader is referred to Chu et al [CHU87c] for further implementation details.

Step 1.

Relocate processes from the longest wait processor to the shortest wait processor. By using processor utilization information and process waiting/queueing time, processors may be identified which have the longest and shortest process waiting times.

Step 2.

Replicate further processes on the shortest waiting processor. This attempts to balance the processing workload by replicating certain processes on the shortest waiting processor.

Step 3.

Delete processes from the longest waiting processor. This is done to reduce inter-processor communication and to further balance the workload.

Thus process replications may improve system load balancing, response time and system reliability and availability. In most cases the algorithm generates very efficient solutions. A factor which has made the implementation of the above two techniques of process placement out of bounds for this research, is that the computation time involved is that of a few minutes on the VAX 11/780. Due to the lack of equipment, research of this kind was not feasible.

11.3. The Dynamic Placement Problem

Static placement involves the placement of a process to a particular processor where it is assumed to remain until execution is completed. In this situation, the costs of a particular allocation are merely the net execution and communication costs for the duration of the program.

It makes sense to change placements dynamically to take advantage of local behaviour of programs and the relatively infrequent changes in program locality [HOR88] [REE84] [SMI88]. To solve the dynamic placement problem, we essentially have to solve a maximum flow problem as the working set of a program changes. In dynamic placement, the cost of reassignment cannot be ignored.

Stone and Bokhari [STO77] base the development of a mathematical model for dynamic placement on the concept of the phase of a modular program. They define the phase as a contiguous period of time during which only one process executes. Each phase is associated with the following information:

1. Which process executes during the phase,
2. Run costs of this process for either of the two processors (in a 2 processor system),
3. Cost of residence of the remaining processes on each of the two processors,
4. Inter-process communication costs between executing the currently executing process and all other processes assigned to different processors, and
5. Relocation cost for each process - the cost of relocating each process from one processor to the other, if relocation were to be carried out at the end of the current phase.

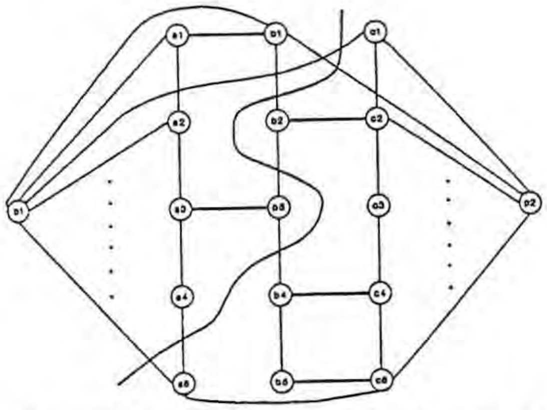


Figure 27 - the result of dynamic placement of a group of processes

Figure 27 shows how the information may be represented by a graph. The number of nodes in the graph equals the number of processes multiplied by the number of phases. The nodes are arranged in a grid with the vertical columns of nodes representing the processes, and the horizontal rows representing the phases. The horizontal edges represent communication costs and the vertical edges represent relocation costs. Two nodes are added, marked P_1 and P_2 which represent the two processors. Edges representing the run costs are drawn from P_1 and P_2 to each of the nodes representing executing processes. Edges representing residence costs are drawn from P_1 and P_2 to the remaining (non-executing) nodes.

As in static placement using graph theory, the minimum weight cut gives the optimal dynamic placement. The graph theory algorithm is similar to that described for static placement. Details may be found in [STO77].

The dynamic problem is no harder that the static one as far as the calculations of maximum flow are concerned, but the difficulty lies in detecting a change in the working set of a program. Since a control program must intervene to perform inter-process transfers across processor boundaries, it should monitor such transfers, and use changes in the rate and nature of such transfers as a signal that the working set has changed.

The dynamic placement problem is very closely related to that of *dynamic load balancing*. Most research on this area involves a certain amount of queueing theory, and Chow and Kohler [CHO79] discuss a job routing strategy which is designed to reduce the average turnaround time of the processes by balancing the total load among the processors. An arriving job is routed by a job dispatcher to one of m parallel processors. The next processor is chosen to minimize or maximize the expected value of a performance related criterion function, which is proportional to the processing power of the processor.

Each processor is modeled as a queue/server model. Processor i is assumed to be characterized completely by its mean service rate u_i. Jobs enter the system at mean rate A. The minimum response time policy routes an arriving job to the system that offers the least expected turnaround time. The minimum system time policy is motivated by two goals, to find an optimal routing strategy and to model parallel processing. The minimum response time and minimum system time policies are functions of the system queue lengths and processor service rate only. The job dispatcher may also have information available on the job arrival rate. The state of the system is a snapshot of the workload distribution among the system components. Figure 28 graphically depicts this situation where n_i is the number of jobs in the i^{th} queue.

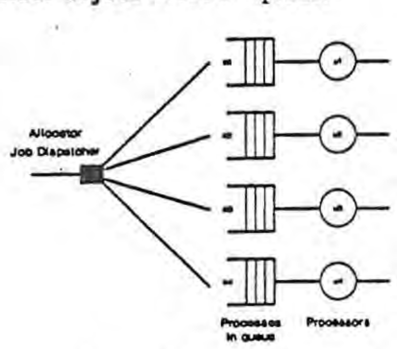


Figure 28 - a queueing model of processes and processors

The literature suggests that two-processor systems have been developed in which program processes may float from processor to processor at load time or during the execution of the program. The ability to reassign program processes to different processors in a distributed system is essential to make the best use of system resources as programs change from one computation phase to another, or as system load changes [STO77]. The migration of processes is not necessarily practical in a hard real-time system, because the migration is so time consuming. The solution is to duplicate processes, so that the same process code occurs on each processor. State variables, related to each process, are constantly updated and passed among the processors, so that a processor can decide whether it is more suited to be executing a particular process.

12. Related Research

Although some of the test results given here may make the algorithms appear impractical, the algorithms are all being used in varying implementations by researchers and scientists. The idea behind the integer programming method has been used by Chu et al [CHU87b] to obtain effective placements within a Defense system known as *The Distributed Processing Architecture Design System*. Here an exhaustive search of all leaves within the search tree is executed to obtain a minimum-bottleneck placement.

A combined network has been implemented at the IBM Watson Research Center, USA [PAR88]. A combined network is one which makes use of both shared and message passing communication protocols. The method of process placement is that of self-scheduling at run-time, where each process that needs to execute decides on which processor from those available which satisfy the necessary memory and communication requirements, it wishes to execute.

The CONVEX system developed at the Convex Computer Corporation, USA, uses dynamic process placement [PAR88]. Instead of processes being allocated to processors, the processors are self-allocating, i.e. each process looks through a ready queue of processes to see if there is work to be done. Each processor selects a process from those processes that need to run, according to its own capabilities.

The RMIT/CSIRO Parallel Systems Architecture project is a joint collaboration project between the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organization [ABR88]. The purpose of the project is to investigate parallel algorithms, methodologies, languages and architectures. The system used is an MIMD one, but with a dataflow implementation as opposed to message passing or shared memory. The allocation problem here is how to distribute the data-flow graph over the available processing elements. It has been realised that to be able to maximize the available parallelism in the system, the information needed must be obtained while the graph is being executed. This is a complex task and consequently, the RMIT machine allocates nodes using a uniform random distribution algorithm. This allocation is said to produce 80% optimal (*sic*) allocations. Further, to allow code blocks to be executed repeatedly in parallel, a hashing algorithm has been developed which hashes a particular value each time a code block is invoked, in order to determine on which machine the code must run.

13. Conclusion

Three methods of static allocation of processes to processors within the abstracted version of occam have been suggested. These are integer programming, graph theoretic and heuristic methods. The suite of test programs which were used to compare the allocation methods were contrived to exhibit a reasonably coarse grain of parallelism, since this was the class of program for which the target network was known to be a reasonable support environment. In spite of the contrived nature of the test programs, the application of the same set of tests to each of the allocation methods enabled a reasonable comparative study of the methods to be made.

The results of experiments performed confirm that optimization techniques of mathematical programming are too slow to be practically used in workload partitioning problems. Obtaining strictly optimal solutions is essentially an academic issue, for the optimization model itself is only a crude approximation of the real-world. The reduction in execution time of the graph theoretic algorithm is very impressive. However, the problem of n-processor systems soon becomes computationally too complex. This, together with the fact that it is difficult to include all the necessary constraints for process allocation into the algorithm, make it not viable for real-time systems. It would seem that, although the heuristic approach yields unsatisfactory results in the test runs carried out, investigations into this method show that there is scope to extend and improve it, and it is felt that with further research, a solution to the problem of process placement may well be found in this area.

Static allocation may result in too much idle processor time because a processor is *booked* by a particular process and no other process can use that processor until that process has finished executing on it. Further, more information which could be used by the allocation algorithm, e.g. processing power of available processors, e.g. AT/PC, should be placed in the NETWORK MAP.

Static allocation cannot stand alone as the solution to the problem of the allocation of processes to processors. The generation of statistical information is required to render the allocation useful for real-time applications, but this is not ideal as the gathering of statistics extends the execution time of the distributed application program, even though only by 10%. Inter-process communication has the greatest influence on the allocation of processes.

It must be noted that to distribute processes automatically, the implementation language, in this case occam, has restrictions imposed upon it: no global variable declarations may be shared and no processes may be created dynamically. For this reason standard occam is amended, although this is not a healthy approach to developing the tools needed for distributed processing. It would be better if languages were designed with the initial purpose of serving distributed programming. If not, then at least established languages such as Modula-2, should be extended, as it is only in these languages that real-time systems have been developed which can be used for more effective research. More importantly, workload partitioning should form an integral step of a computerized system design process, the main reason being that workload partitioning itself is an iterative process which heavily depends on model data, and which continues throughout the system development cycle.

The goal of automatic allocation is an ideal one and still requires a great deal of research, yet it must be persued if concurrent software is to be truly portable. True portability implies that the user has the ability to retarget the software via an entirely mechanical process. The user should not be concerned with the structure of the hardware, nor need to make any direct or indirect changes to the source code. This issue will become more pertinent as more concurrent programming languages are developed and simulated transputer networks are replaced by actual transputer topologies.

Of the three static process allocation techniques implemented and tested, the heuristic approach holds the most potential for further research in the field of process placement in distributed systems. This is due to the algorithm's promising performance in execution time, as opposed to the very poor execution speed of the integer programming method, and the constraint limitations of the graph theoretic approach. Faster and more reliable techniques still remain to be established.

References

[ABR88] Abramson, D.A., Egan, G.K. (1988) *An Overview of the RMIT/CSIRO Parallel Systems Architecture Project*, The Australian Computer Journal, Vol. 20, No. 3, 113-121.

[AND73] Anderberg, M.R. (1973) *Cluster Analysis For Applications*, Academic Press, Inc., New York.

[AND88] Andriessen, J.H.M. (1988) *Task Scheduling Programming System for the Delft Parallel Processor*, Microprocessing and Microprogramming, Vol. 23, No's. 1-5, 283-288.

[ARO81] Arora, R.K., Rana, S.P., Sharma, N.K. (1981) *On the Design of Process Assigner for Distributed Computing Systems*, The Australian Computer Journal, Vol. 13, No. 3, 77-82.

[BEC64] Beckenbach, E.F. ed. (1964) *Applied Combinatorial Mathematics*, John Wiley and Sons, Inc., New York.

[BIS87a] Bishop, J.M. (1987) *Ada for multi-microprocessors: some problems and solutions*, Computer Science Department , University of the Witwatersrand.

[BIS87b] Bishop, J.M., Adams, S.R., Prichard, D.J. (1987) *Distributing concurrent Ada programs by source translation*, Software - Practice and Experience, Vol. 17, No. 12, 859-884.

[BOW88] Bowler, M.C., Morse, M.J., Frydas, N. (1988) *Adaptive Routing in Simulated Computer Networks*, Developments using occam, Occam Users Group, Ed. Kerridge, J., IOS Publishers, Amsterdam, The Netherlands, 137-141.

[BRO86] Brookes, G.R., Manson, G.A., Thompson, J.A. (1986) *Lattice and ring array topologies using transputers*, Computer Communications, Vol. 9, No. 3, 121-125.

[BUR88a] Burns, A. (1988) *Programming in occam 2*, Addison-Wesley Publishing Company, Inc., Great Britain.

[BUR88b] Burton, F.W., McKeown, G.P., Rayward-Smith, V.J. (1988) *On Process Assignment in Parallel Computing*, Information Processing Letters, Vol. 29, 31-34.

[CHA88] Chandy, K.M., Misra, J. (1988) *Parallel Program Design A Foundation*, Addison-Wesley Publishing Company, Inc., New York.

[CHE80] Chen, P.P., Akoka, J. (1980) *Optimal Design of Distributed Information Systems*, IEEE Transactions on Computers, Vol. c-29, No. 12, 1068-1080.

[CHO79] Chow, Y-C, Kohler, W.H. (1979) *Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System*, IEEE Transactions on Computers, Vol. c-28, No. 5, 354-361.

[CHU80] Chu, W.W., Holloway, L.J., Lan M., Efe, K. (1980) *Task Allocation in Distributed Data Processing*, Computer, Vol. 13, No. 11, 57-69.

[CHU87a] Chu, W.W., Kim, K.H., McDonald, W.C. (1987) *Testbed-Based Validation of Design Techniques for Reliable Distributed Real-Time Systems*, Proceedings of the IEEE, Vol. 75, No. 5, 649-667.

[CHU87b] Chu, W.W., Lan, L.M-T. (1987) *Task Allocation and Precedence Relations for Distributed Real-Time Systems*, IEEE Transactions on Computers, Vol. c-36, No. 6, 667-679.

[CHU87c] Chu, W.W., Leung, K.K. (1987) *Module Replication and Assignment for Real-Time Distributed Processing Systems (Invited Paper)*, Proceedings of the IEEE, Vol. 75, No. 5, 547-562.

[CHU87d] Chu, W.W., Sit, C-M. (1987) *A Batch Service Scheduling Algorithm with Time-Out for Real-Time Distributed Processing Systems*, Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, September 21-25, 1987, 250-257.

[CLA88] Clayton, P.G., Handler, C., Hill, D.T. (1988) *Abstract Process Placement for Distributed Parallel Processing Environments*, Proceedings of International Parallel Processing Symposium, Johannesburg, South Africa.

[COO74] Cooper, L., Steinberg, D. (1974) *Methods and Applications of Linear Programming*, W.B. Saunders Company, Philadelphia.

[COO80] Cook, R.P. (1980) **MOD - A Language for Distributed Programming*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, 563-571.

[DUR74] Duran, B.S., Odell, P.L. (1974) *Lecture Notes in Economics and Mathematical Systems - Cluster Analysis, A Survey*, No. 100, Springer-Verlag, Berlin.

[EFE82] Efe, K. (1982) *Heuristic Models of Task Assignment Scheduling in Distributed Systems*, IEEE Computer, Vol. 15, 50-56.

[ELD80] El-Dessouki, O.I., Huen, W.H. (1980) *Distributed Enumeration on Between Computers*, IEEE Transactions on Computers, Vol. c29, No. 9, 818-825.

[EVE74] Everitt, B. (1974) *Cluster Analysis*, 2nd Edition, Halsted Press, Division of John Wiley & Sons, New York.

[FIS86] Fisher, A.J. (1986) *A Multi-processor Implementation of occam*, Software Practice and Experience, Vol. 16, No. 10, 875-892.

[GAJ85] Gajski, D.D., Peir, J. (1985) *Essential Issues in Multiprocessor Systems*, Computer, Vol. 18, No. 6, 9-27.

[GAR87] Garnett, N., King, J., Veer, B. (1987) *Helios Technical Manual*, Perihelion Software Limited, 1-3.

[GON80] Gonzalez, M.J., Jordan, B.W. (1980) *A Framework for the Quantitative Evaluation of Distributed Computer Systems*, IEEE Transactions on Computers, Vol. c-29, No. 12, 1087-1094.

[GOU88] Gough, K.J., Mohay, G.M. (1988) *Modula-2 A Second Course in Programming*, Prentice-Hall Inc., Australia.

[GYL76] Gylys, V.B., Edwards, J.A. (1976) *Optimal Partitioning of Workload for Distributed Systems*, Digest of Papers, IEEE Computer Society COMPCON '76 Proceedings, 353-357.

[HAN78] Hansen, B. (1978) *Distributed Processes: A Concurrent Programming Concept*, Communications of the ACM, Vol. 21, No. 11, 934-941.

[HIL86] Hill, D.T. (1986) *Simulating a Transputer*, Computer Science Honours Project, Department of Computer Science, Rhodes University, Grahamstown.

[HIL88] Hill, D.T. (1988) *Towards a Portable occam*, MSc. (Applied Computer Science) Thesis, Department of Computer Science, Rhodes University, Grahamstown.

[HOR88] Horton, I.A., Turner, S.J. (1988) *Dynamic Processes in occam*, Developments using occam, Occam Users Group, Ed. Kerridge, J., IOS Publishers, Amsterdam, The Netherlands, 11-21.

[HUI87] Huijsman, R.D., van Katwijk, J., Toetenel, W.J. (1987) *Performance Aspects of Ada Tasking in Embedded Systems*, Microprocessing and Microprogramming, The Euromicro Journal, Vol. 21, No's. 1-5, 301-310.

[HUL85] Hull, M.E.C. (1985) *Implementations of the CSP Notation for Concurrent Systems*, The Computer Journal, Vol. 29, No. 6, 500-505.

[KAR80] Kartashev, S.P., Kartashev, S.I. (1980) *Supersystems for the 80's*, Computer, Vol. 13, No. 11, 11-14.

[INM87] Inmos (1987) *The transputer instruction set - a compiler writer's guide*, INMOS Limited, Bristol.

[KAS84] Kasahara, H., Narita, S. (1984) *Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing*, IEEE Transactions on Computers, Vol. c-33, No. 11, 1023-1029.

[LAK84] Lakhani, G.D. (1984) *An improved Distribution Algorithm for Shortest Paths Problem*, IEEE Transactions on Computers, Vol. c-33, No. 9, 855-857.

[LAM84] Lambert, J.E., Halsall, F. (1984) *Program debugging and performance evaluation aids for a multi-microprocessor development system*, Software and Microsystems, Vol. 3, No. 1, 2-10.

[LIU73] Liu, C.L., Layland, J.W. (1973) *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the Association fro Computing Machinery, Vol. 20, No. 1, 46-61.

[MAK73] Maki, D.P., Thompson, M (1973) *Mathematical Models and Applications*, Prentice-Hall, Inc., Englewood Cliffs.

[MAP82] Ma, P.R. (1982) *A Task Allocation Model for Distributed Computing Systems*, IEEE Transactions on Computers, Vol. c-31, No. 1, 41-47.

[MAY88] May, D., Shepherd, R. (1988) *Current and Future Transputers*, INMOS Ltd, Bristol, 1-5.

[MEL86] Mellor, P.V., Dubery, J.M., Whitehead, D.G. (1986) *Adapting Modula-2 for distributed systems*, Software Engineering Journal, September, 184-189.

[MUR88] Murray, K.A., Wellings, A.J. (1988) *Issues in the Design and Implementation of a Distributed Operating System for a Network of Transputers*, Microprocessing and Microprogramming, Vol. 24, 169-177.

[NEW86] Newport, J.R. (1986) *An introduction to Occam and the development of parallel software*, Software Engineering Journal, July, 165-168.

[OBE88] Obermeier, K.K. (1988) *Side by Side*, Byte, Vol. 13, No. 11, 275-283.

[PAR88] *Parallel Processing '88*, Symposium Record, Part II, Johannesburg, South Africa, 26-28 October 1988.

[POU86] Pountain, D. (1986) *Personal Supercomputers*, Byte, Vol. 11, No. 7, 363-368.

[POU88] Pountain, D, May, D. (1988) *A tutorial introdution to occam 2*, Inmos Limited.

[QUI87] Quinn, M.J. (1987) *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Book Company

[RAT87] Ratheal, S., Lombardi, F. (1987) *A Software Testbed for the Design and Evaluation of Distributed Computer Systems*, Microprocessing and Microprogramming, Vol. 19, 49-58.

[REE84] Reed, D.A. (1984) *The Performance of Multimicrocomputer Networks Supporting Dynamic Workloads*, IEEE Transactions on Computers, Vol. c-33, No. 11, 1045-1048.

[SAN86] Sanguinetti, J. (1986) *Performance of a Message-Based Multiprocessor*, Computer, Vol. 19, No. 9,47-55.

[SCH87] Schrott, G. (1987) *A Generalised Task Concept for Multiprocessor Real-Time Systems*, Microprocessing and Microprogramming, The Euromicro Journal, Vol. 20, No's. 1-3, 85-90.

[SHA87] Sharp, J.A. (1987) *An Introduction to Distributed and Parallel Processing*, Blackwell Scientific Publications, Oxford.

[SHE86] Sherman, M., Marks, A. (1986) *Using Low-Cost Workstations to investigate Computer Networks and Distributed Systems*, Computer, Vol. 19, No. 6, 32-41.

[SHI81] Shiloach, Y., Vishkin, U. (1981), *An $O(n^2 \log n)$ Parallel MAX-FLOW Algorithm*, Journal of Algorithms, No. 3, 128-146.

[SMI88] Smith, J.M. (1988) *A Survey of Process Migration Mechanisms*, ACM SIGPLAN Notices, Vol. 23, No. 3, 28-40.

[STE88] Stein, R.M. (1988) *T800 and Counting*, Byte, Vol. 13, No. 11, 287-296.

[STO77] Stone, H.H. (1977) *Multiprocessor Scheduling with the Aid of Network Flow Algorithms*, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, 85-93.

[TAN81] Tanenbaum, A.S. (1981) *Computer Networks*, Prentice-Hall Inc., Englewood Cliffs.

[TRY70] Tryon, R.C., Bailey, D.E. (1970) *Cluster Analysis*, McGraw-Hill, Inc., New York.

[VAN77] Van Ryzin, J. (ed.) (1977) *Classification and Clustering*, Academic Press, Inc., New York.

[VAN81] Van Tilborg, A.M., Wittie, L.D. (1981) *Distributed task force scheduling in multi-microcomputer networks*, National Computer Conference 1981, 283-289.

[WAH84] Wah, B.W. (1984) *A Comparative Study of Distributed Resource Sharing on Multiprocessors*, IEEE Transactions on Computers, Vol. c-33, No. 8, 700-711.

[WAL85] Walker, P. (1985) *The Transputer*, Byte, Vol. 10, No. 5, 219-235.

[WIL88] Wilson, P (1988) *Parallel Processing Comes to PC's*, Byte, Vol. 13, No. 11, 213-218.