ABSTRACT.

HEURISTIC ALGORITHMS FOR DISTRIBUTED PROCESSOR SCHEDULING WITH LIMITED MEMORY.

by

Timothy A. Gonsalves.

This thesis studies a heuristic approach to scheduling on a 2-processor distributed system when one processor has a limited memory. A program is assumed to consist of a set of sequentially executing modules and is represented by Stone's graph model. It is desired to assign these modules to the processors so as to minimize interprocessor communication while taking advantage of specific capabilities of the two processors. This optimization problem is NP-complete. It is shown that the corresponding 'absolute approximation' problem is as hard. For the classes of constant degree and constant connectivity graphs statistics are presented to support the conjecture of Rao, Stone and Hu that use of the 'inclusive-cuts graph' can appreciably simplify this scheduling problem. Asymptotic upper and lower bounds on the expected cost of the optimum assignment are derived for the class of constant degree graphs.

These results motivate the development and assist the evaluation of two polynomial-time heuristic algorithms for
2-processor scheduling with limited memory. For constant degree graphs it is shown that the heuristics can be useful scheduling tools. It is also shown that use of the inclusive-cuts graph can lead to an improvement in performance, but at the expense of additional scheduling overhead.

An ancillary result proved is a relationship between scheduling with limited memory and scheduling when one processor is multiprogrammed and has a variable load factor. Some implications of this relationship are discussed.
ACKNOWLEDGEMENTS.

I sincerely thank Dr. G.S.Rao for his continual guidance, encouragement and advice in this work and all aspects of graduate study. My thanks to Dr. E.A.Feustel and Dr.J.R.Jump for serving on my thesis committee. A special word of thanks to Bart Sinclair for many fruitful discussions. To Daniel, Anand, Pierre, William, Naresh and Sid thanks for making my stay at Rice a pleasurable experience.

This work was supported by the National Science Foundation principally under Grant # MCS-77-04364 and also under Grants # MCS-72-0351 and # MCS-76-07683.
REFLECTIONS ON INGENUITY

Here's a good rule of thumb:
Too clever is dumb.

- Ogden Nash
# Table of Contents

**CHAPTER 1**  INTRODUCTION  1

**CHAPTER 2**  2-PROCESSOR SCHEDULING  6  
  2.1  2-processor scheduling without constraints  6 
  2.2  Scheduling with limited memory  15 
  2.3  Scheduling with variable load factor  23 
  2.4  Summary  25 

**CHAPTER 3**  A STUDY OF LIMITED MEMORY SCHEDULING  26  
  3.1  Complexity of approximation algorithms for limited memory scheduling  26 
  3.2  Node condensation in the ICG  30 
  3.3  Asymptotic analysis of limited memory scheduling  37 
  3.4  Summary  50 

**CHAPTER 4**  HEURISTICS FOR LIMITED MEMORY SCHEDULING  52  
  4.1  A heuristic based on the ICG  52 
  4.2  A second heuristic  64 
  4.3  Results of simulation  72 
  4.4  Summary  84 

**CHAPTER 5**  CRITICAL LOAD FACTORS AND THE ICG  85  
  5.1  Summary  92 

**CHAPTER 6**  CONCLUSION  93  
  6.1  Summary  93
6.2 Suggestions for further work

REFERENCES
CHAPTER 1. INTRODUCTION.

A distributed processor system consists of several autonomous processing elements physically and logically interconnected to form a single system. The processing elements cooperate in the execution of user programs. A system-wide executive handles resource allocation and process scheduling in such a manner that the distributed nature of the system is transparent to the user. The user has access to system resources without being aware of the physical location of these resources (Agarwala [1976], Enslow [1978], Jensen [1978]).

The principal motivation for distributed computer systems is the potential for enhanced flexibility, integrity and performance. Flexibility arises from the autonomy of system units and the transparency of the configuration to the user. Hardware modules can be added or removed without software changes. Integrity covers tolerance to total failure of modules as well as to intermittent errors. Failing processors can be dynamically removed from the system without affecting user programs with the possible exception of the program in execution on the processor in question. Tolerance to intermittent errors can be improved by replicating processes on different processors and using majority voting. Performance is usually measured by throughput and turnaround time. Throughput is increased
by the multiplicity of processors. Turnaround time can be reduced by load balancing and by providing a variety of special-purpose functional units that would be uneconomical in a single centralized system.

A category of distributed systems that is of considerable interest is that of 2-processor systems. Typically a minicomputer is connected to a large multiprogrammed computer by a communication link. The minicomputer is I/O-oriented while the mainframe computer provides processing power and a large memory. Satellite graphics systems such as the ones at Brown University (van Dam et al. [1973]) and North Carolina University (Foley [1976]) fall into this category. Program modules can usually float from processor to processor during program execution. To minimize execution costs modules should be assigned to the processor on which they execute fastest. It is also desirable to have modules in the same working set resident on the same processor to minimize interprocessor communication costs. These two objectives may be conflicting. Thus we desire to find an assignment of modules to processors that minimizes the sum of the execution and communication costs.

Stone [1977] proposes for 2-processor systems a network model which he calls a 'processor-flow graph'. He obtains the solution to the optimum assignment problem using a network-flow algorithm. Rao et al. [1977] consider
the case when one processor has a limited memory. This processor may not be able to accommodate all the modules assigned to it by the optimum assignment, that is, this assignment may not be feasible. It is desired to find the minimum cost feasible assignment. It can be shown that the limited memory scheduling problem is NP-complete. Thus it is not likely that an efficient algorithm exists which finds an optimum solution in all cases. Stone’s processor-flow graph can be transformed into a graph called the 'inclusive-cuts graph'. The two graphs are equivalent with respect to the limited memory scheduling problem. Subsets of nodes in the processor-flow graph may map onto single nodes in the inclusive-cuts graph. Thus the transformation may appreciably reduce the size of the problem and consequently reduce the amount of work necessary to find the optimum solution in the limited memory case.

This thesis examines analytically and by simulation the performance of two polynomial-time heuristic algorithms for finding possibly sub-optimal solutions to the 2-processor limited memory scheduling problem. One heuristic uses the inclusive-cuts graph while the other uses only the processor-flow graph. We provide experimental results on the performance of the heuristics for a class of randomly generated problem instances. For this class of problem instances we show that asymptotically the expected cost of the heuristic solutions is no more than twice that of the optimum solution.
For graphs of up to 82 nodes the observed performance of the heuristics is considerably better than this bound. Thus the heuristics can be viable tools for scheduling. We show that the heuristic which uses the inclusive-cuts graph performs better than the other but is less efficient.

The significance of this work lies in the fact that the problem is of practical importance but is NP-complete. Hence, unless P=NP we may be forced to use such heuristics if we desire to obtain a solution efficiently. Efficiency is necessary since the scheduler is a frequently used component of an operating system. Also assignments may have to be dynamically recomputed if system variables change, thus increasing the usage of the scheduler.

As an interesting corollary to this work we are able to show a connection between scheduling with limited memory and scheduling when one processor is multiprogrammed and has a variable load. The load is measured by the 'load factor' which varies between 0 and 1. Stone [1978] shows that if one processor in a 2-processor system has a variable load factor then modules move in clusters between processors at specific 'critical load factors' which may be predetermined. We show that this clustering is related to the reduction in the inclusive-cuts graph. Thus results for one problem may be applied to the other in certain cases.

In chapter 2 we review prior work on 2-processor schedul-
Chapter 3 contains results on the complexity of the limited memory scheduling problem, the reduction in the inclusive-cuts graph and the asymptotic behavior of a class of heuristic algorithms. The two algorithms which we investigate are described in chapter 4 where we also discuss their performance. In chapter 5 we show the connection between scheduling with variable load factor and scheduling with limited memory. We summarize our work in chapter 6.
CHAPTER 2. 2-PROCESSOR SCHEDULING.

In this chapter we describe the unconstrained 2-processor scheduling problem and review Stone's [1977] network model and solution to the problem. We then outline the work of Rao et al. [1977] on the limited memory scheduling problem. In particular we describe the construction of the inclusive-cuts graph. Finally we describe the results of Stone [1978] on variable load factor scheduling.

2.1 2-PROCESSOR SCHEDULING WITHOUT CONSTRAINTS. (Stone [1977])

A program is assumed to consist of a number of modules. Modules can communicate with each other through a CALL mechanism with parameter passing. Program execution is strictly sequential. Some modules are free to be executed on either processor while others must be assigned to a particular processor. The latter case arises when a module requires resources such as floating-point hardware or a graphics display which are available on only one processor. Each module has a non-negative cost of execution on each processor. If a module cannot be assigned to a processor the corresponding cost is set to infinity. Since the processors are coupled by an intermediate-speed communications link intermodule references between modules on different processors can incur a non-negative communication cost. This cost is assumed to be symmetric i.e. for modules A and B the communication cost is the same whether
A is assigned to processor $P_1$ and B to processor $P_2$ or vice-versa. Inter-module references between modules on the same processor are assumed to have negligible cost. The problem is to find an assignment of modules to processors that minimizes the total cost of execution and communication. There is no limit on the number of modules that can be assigned to either processor.

The costs considered can be a function of any variables of interest to the user such as elapsed time or monetary costs. The costs can be estimates or can be based on previous runs. The assignment is static for the life of the program.

### 2.1.1 THE NETWORK MODEL

Stone models the problem by a processor-flow graph and uses a network-flow algorithm to solve it. First we construct a module interconnection graph. Each module is represented by a node. There is an undirected edge between two nodes if there is communication between the corresponding modules. The weight on the edge is the total cost of communication between the modules if they are assigned to different processors. The table of module communication costs for an example program is given in Table 2.1 and the corresponding module interconnection graph is shown in Fig. 2.1.
Table 2.1 Intermodule Communication Costs.

<table>
<thead>
<tr>
<th>Module Pair</th>
<th>Communication Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B</td>
<td>4</td>
</tr>
<tr>
<td>A, C</td>
<td>2</td>
</tr>
<tr>
<td>B, C</td>
<td>1</td>
</tr>
<tr>
<td>B, D</td>
<td>7</td>
</tr>
<tr>
<td>C, D</td>
<td>7</td>
</tr>
</tbody>
</table>

Fig. 2.1 Module Interconnection Graph.
To complete the processor-flow graph two nodes S and T are added to represent the two processors. (Note: each node is labelled with the name of the processor or module that it represents. When there is any ambiguity we qualify the name eg. processor S or node S). An undirected edge is added between each processor-node and every module-node. The weight on the edge between module-node M and processor-node T is the total execution cost of module M on processor S over the life of the program. Similarly the weight on the edge between nodes M and S is the total execution cost of module M on processor T. Fig. 2.2 shows the processor-flow graph obtained from the module-interconnection graph of Fig. 2.1 and the execution costs in Table 2.2. Note that module B has an infinite cost of execution on processor S and must be assigned to processor T.

We now introduce some necessary graph terminology. A \textbf{cutset} of a graph G is defined as a set of edges the removal of which partitions G into two disjoint subsets. No proper subset of a cutset is itself a cutset. The weight or value of a cutset is the sum of the weights on the edges in the cutset. A \textbf{minimum weight cutset (mincut)} is a cutset with minimum weight among all cutsets in the graph.

Consider any cutset that partitions a processor-flow graph into subsets G₁ and G₂ such that processor-node S is in G₁ and processor-node T is in G₂. By assigning to S
Table 2.2 Module Execution Costs.

<table>
<thead>
<tr>
<th>Module</th>
<th>Execution Cost</th>
<th>Processor S</th>
<th>Processor T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2.2 Processor-Flow Graph.
the modules corresponding to the module-nodes in $G_1$ and the remainder to $T$ we obtain a unique assignment of modules to the two processors. Clearly there is a one-to-one correspondence between cutsets of the processor-flow graph and module assignments.

The cost of an assignment is equal to the weight of the corresponding cutset. For every pair of modules $A$ and $B$ such that $A$ and $B$ are assigned to different processors the edge $AB$ is in the cutset. The weight on edge $AB$ is the communication cost between modules $A$ and $B$ if they are assigned to different processors. Thus communication costs are accounted for. If processor-node $S$ is in subset $G_1$ then for every module $A$ assigned to processor $S$ module-node $A$ is in $G_1$. Since processor-node $T$ is in subset $G_2$ the edge $AT$ is in the cutset. The weight on edge $AT$ is the execution cost of module $A$ on processor $S$. This accounts for execution costs. Thus the problem of finding the minimum cost assignment is equivalent to the problem of finding the minimum weight cutset separating the two processor-nodes in the processor-flow graph.

The minimum weight cutset can be found by using a network flow algorithm initially formulated by Ford and Fulkerson [1962]. The algorithm is used to find the maximum flow in a commodity flow network.
2.1.2 **COMMODITY FLOW NETWORKS.**

A commodity flow network consists of a set of source nodes \( \{T_i\} \), a set of sink nodes \( \{S_i\} \) and a set of interior nodes \( \{N_i\} \). The source nodes are linked to the sink nodes possibly through the interior nodes by undirected weighted edges. The source nodes represent production centers each with infinite output. The sink nodes represent demand centers each capable of absorbing an infinite amount of commodity. The edges represent transportation links. The weight on an edge represents the amount of commodity that can be transported along that link. Interior nodes neither produce nor consume commodity. Fig. 2.3a shows a commodity flow network.

A commodity flow is represented by directed, weighted arrows along the edges. The weight represents the quantity of commodity flow along the edge in the direction of the arrow. A feasible commodity flow satisfies the following constraints:-

(i) there is a net non-negative flow out of each source node and into each sink node,
(ii) the net flow into (or out of) each interior node is zero,
(iii) the flow along any edge is less than or equal to the weight on that edge.

Fig. 2.3b shows a feasible commodity flow for the network of Fig. 2.3a.
Fig. 2.3a  Commodity-flow network showing capacities.

Fig. 2.3b  Commodity-flow network showing a feasible flow.
The value of a commodity flow is the total flow out of all the source nodes. This is equal to the total flow into all the sink nodes. A maximum flow is a feasible commodity flow the value of which is maximum among all feasible flows.

The algorithm of Ford and Fulkerson finds a maximum flow in a commodity flow network. As a consequence of the following theorem it also finds a minimum cutset separating the source nodes from the sink nodes. 

**Max-flow, Min-cut Theorem** (Ford and Fulkerson [1962]).

This theorem states:

"The value of a maximum flow in a commodity network is equal to the weight of a minimum cutset separating the source and sink nodes in the network."

The max-flow algorithm of Ford and Fulkerson has a running time that can be arbitrarily large in certain cases. Edmonds and Karp [1972] modified it to have a worst-case time complexity of $O(n^5)$, where $n$ is the number of nodes in the network. Dinic [1970] improved the performance to $O(n^4)$ and Karzanov [1974] modified Dinic's algorithm to achieve a bound of $O(n^3)$. Even [1976] presents an exposition of the Dinic-Karzanov algorithm.

**2.1.3 SOLUTION OF THE SCHEDULING PROBLEM.**

Clearly, a processor-flow graph is a commodity flow
graph if we treat one processor as the source and the other as the sink. The communication and execution costs on the edges now represent transportation link capacities. Applying a max-flow algorithm to the processor-flow graph, as a result of the max-flow, min-cut theorem we obtain a mincut between the processor-nodes. As described earlier this defines an assignment of modules to the two processors. This assignment is a minimum cost solution to the scheduling problem.

We have described the unconstrained 2-processor scheduling problem and have given Stone's network model and solution procedure using a network flow algorithm. This procedure is used in the constrained scheduling problems which we describe in the remainder of this chapter.

2.2 SCHEDULING WITH LIMITED MEMORY. (Rao et al.,[1977])

In this section we introduce the constraint that one processor, say T has a limited memory capacity while the other, S has an infinite memory. The size of each program module is known. A feasible assignment is one in which the total size of the modules assigned to T is less than or equal to its memory capacity. If the minimum cost assignment is not feasible we want to find the minimum cost feasible assignment.

Since the 2-processor limited memory scheduling problem
is NP-complete it is unlikely that there exists a polynomial-time algorithm for the problem. An exponential-time algorithm is computationally practicable only for small instances of the problem. Rao et al. reduce a processor-flow graph to an inclusive-cuts graph (ICG). The two are shown to be equivalent with respect to the minimum feasible cut. The size of the ICG can be appreciably smaller than that of the processor-flow graph. In section 3.1 we present some statistical evidence for this. In addition there is a partial ordering on the nodes in the ICG. This ordering and the reduction in size can greatly reduce the number of cuts to be examined in an exhaustive search for the minimum feasible cut. Thus construction of the ICG can substantially increase the domain of practicably solvable instances of the problem. In chapter 4 we examine the performance of two similar heuristic algorithms for the limited memory scheduling problem. One involves the use of the ICG while the other does not. The results of our simulation indicate that the use of the ICG leads to an improvement in performance.

2.2.1 **THE INCLUSIVE-CUTS GRAPH (ICG).**

We describe the algorithm for construction of the ICG of a graph and give an example. Formal proofs can be found in the paper referenced.
Step 1.
Find the mincut separating the two processors, S and T in the given processor-flow graph. To find the minimum feasible assignment we need only reassign nodes from T to S and never from S to T. Thus S and the nodes assigned to S by the mincut can be condensed into a single node which we also call S. Fig. 2.4a shows a processor-flow graph and Fig. 2.4b shows the reduced graph after node A is condensed into S.

Step 2.
For each node M in the reduced graph find $M(S)$, the mincut between S and T that reassigns M to S. $M(S)$ can be obtained by application of the maxflow, mincut algorithm to the reduced graph obtained in step 1 with T as the source and M and S as sinks. Equivalently, set the weight on the edge between M and S to infinity and find the mincut between S and T. Note that the mincut found when M and S are sinks does not include the edge MS. Thus increasing the weight on this edge to infinity does not change the value of the mincut. Hence the two methods described for finding $M(S)$ are equivalent. Fig. 2.4c shows the mincuts reassigning each node to S.

Step 3.
Find the disjoint subsets $S_M$ such that for every node $N \in S_M$ $N(S)$ is the same. This implies that if any node
Fig. 2.4a Processor-flow Graph.

Fig. 2.4b Reduced graph after condensing A into S.
Fig. 2.4c Cuts reassigning each node to S.

Table 2.3 Subsets reassigned to S by mincuts.

<table>
<thead>
<tr>
<th>Condensed Node, M</th>
<th>Subset of Nodes in $S_M$</th>
<th>Condensed nodes assigned to S by $M(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>{B}</td>
<td>{B}</td>
</tr>
<tr>
<td>C</td>
<td>{C}</td>
<td>{C}</td>
</tr>
<tr>
<td>D'</td>
<td>{D, E, F}</td>
<td>{D', B, C}</td>
</tr>
<tr>
<td>G</td>
<td>{G}</td>
<td>{G, D', B, C}</td>
</tr>
</tbody>
</table>
in $S_M$ is reassigned to $S$ then every node in $S_M$ is reassigned to $S$. Thus in the original processor-flow graph $S_M$ can be condensed into a single node for the purpose of finding the minimum feasible cut. The remainder of the algorithm uses these condensed nodes. Table 2.3 shows the condensable subsets. In this example nodes D, E and F are condensed into a single node D'. For each condensed node M the table also shows the condensed nodes assigned to S by M(S).

**Step 4.**

Arrange the cuts found in step 2 in increasing order of the number of condensed nodes assigned to S. In case of ties order the cuts by the total memory requirements of modules assigned to S. Further ties are broken arbitrarily. Table 2.3 shows this ordering.

**Step 5.**

Include in the ICG one node for S and one node for T. Add a node for each condensed node identified in step 3. Associate with S the cost of the mincut. With each node M associate the cost of the mincut reassigning M to S.

The remaining steps consider the mincuts in the order established in step 4.

**Step 6.**

For each cut M(S) that reassigns exactly one condensed node to S add an edge to the ICG directed from M to S. Thus in the example we add edges BS and CS to the ICG (Fig.
Step 7.

Select the next cut \( \mathbf{M}(S) \) not yet considered. For each condensed node \( N \) assigned to \( S \) by \( \mathbf{M}(S) \) add an edge from \( M \) to \( N \) in the ICG if and only if \( N \) has no edge incident on it from another node assigned to \( S \) by \( \mathbf{M}(S) \). Repeat step 7 until all cuts have been considered.

For any \( N \) assigned to \( S \) by \( \mathbf{M}(S) \) the set of nodes assigned to \( S \) by \( \mathbf{N}(S) \) is a proper subset of the set assigned to \( S \) by \( \mathbf{M}(S) \). Thus the cut \( \mathbf{N}(S) \) for every node assigned to \( S \) by \( \mathbf{M}(S) \) is considered before \( \mathbf{M}(S) \). There is therefore a directed path from every node \( M \) in the ICG to \( S \). Further, a node is on a path from \( M \) to \( S \) if and only if it is assigned to \( S \) by \( \mathbf{M}(S) \). Thus any cut in the ICG that assigns \( M \) to \( S \) also assigns to \( S \) every node assigned to \( S \) by \( \mathbf{M}(S) \).

In the example \( \mathbf{D}'(S) \) assigns \( B, C \) and \( D' \) to \( S \). Thus we add the edges \( D'B \) and \( D'C \) to the ICG. \( \mathbf{G}(S) \) assigns \( B, C, D' \) and \( G \) to \( S \). Since \( B \) and \( C \) have edges incident on them from \( D' \) we do not add edges \( GB \) and \( GC \). However, we do include the edge \( GD' \).

Step 8.

For each node \( M \) in the ICG that has no edge incident on it add an edge from \( T \) to \( M \).
Fig. 2.4d Inclusive-Cuts Graph.
Fig. 2.4d shows the ICG constructed by the above algorithm from the processor-flow graph of Fig. 2.4a.

2.2.2 APPLICATION OF THE ICG.

The minimum feasible cut is a cut in the ICG (Rao et al. [1977]). The weight of any cut in the ICG is the weight of the corresponding cut in the processor-flow graph. Thus to solve the scheduling problem we first find the ICG of the given processor-flow graph. We then enumerate all feasible cuts in the ICG and take the minimum valued of these as the solution.

In chapter 3 we present experimental results on the expected size of the ICG for two classes of randomly generated graphs. In chapter 4 we examine a heuristic which examines some, but not necessarily all feasible cuts in the ICG to obtain a solution to the limited memory scheduling problem.

2.3 SCHEDULING WITH VARIABLE LOAD FACTOR. (Stone [1978])

In this section we review Stone's work on scheduling when one processor has a variable load factor. Stone shows that as load factor varies clusters of modules move between processors. Later we show a connection between these clusters and the condensable subsets identified by the ICG.
The load factor \( f \) of a processor with respect to a program is defined as the fraction of CPU time devoted to the program. On a multiprogrammed processor the load factor can vary with time between 0, when the program is not running and 1, when the program is the only one running. The total elapsed time for execution of a program, the effective execution time increases as load factor decreases. Studies indicate that for some systems the effective execution time is proportional to \( 1/f \) (Burns [1977], Mamrack [1977]).

In a two-processor system assume that one processor, \( T \) has a variable load factor while the other, \( S \) has a constant load factor. There are no other constraints. If the execution costs of interest are a function of effective execution time then the execution costs of modules on \( T \) vary with the load factor of \( T \). Thus the minimum cost assignment may vary, necessitating a re-assignment of modules. Stone shows that as \( f \) decreases from 1 to 0 modules initially assigned to \( T \) may migrate to \( S \) but modules initially assigned to \( S \) do not migrate. Each module \( M \) initially assigned to \( T \) migrates to \( S \) at a specific load factor, the critical load factor \( f_M \) of the module. The minimum cost assignment assigns \( M \) to \( T \) for \( f > f_M \) and to \( S \) for \( f < f_M \). At \( f = f_M \) the mincut is not affected by the assignment of \( M \). Thus we can compute the critical load factors prior to execution and during program
execution use a simple table-lookup to determine which modules, if any, should be reassigned as load factor varies. Michel and van Dam [1977] present an algorithm for computing critical load factors. The algorithm uses at most \( n \) applications of the maxflow, mincut algorithm to determine the critical load factors for all \( n \) modules of the program.

Several modules may have the same critical load factor. They are reassigned together as load factor varies. Hence for scheduling with variable load factor each such cluster of modules can be treated as a single 'super-module'. In a test program of 50 modules Michel and van Dam found that the modules broke up into relatively few clusters (Stone [1978]). In chapter 5 we show that the modules in a condensable subset of the ICG have the same value of critical load factor and we discuss the implications of this result.

2.4 **SUMMARY.**

In this chapter we have presented an overview of prior work on 2-processor scheduling - unconstrained, with limited memory and with variable load factor. We have described in some detail the network model of the problem and the construction of the inclusive-cuts graph for limited memory scheduling. We have introduced the concept of critical load factors.
CHAPTER 3. A STUDY OF LIMITED MEMORY SCHEDULING.

In this chapter we examine some issues in limited memory scheduling. This motivates our development of heuristics and provides the background for the discussion on their performance in the next chapter. First we prove a result on the complexity of approximation algorithms for limited memory scheduling. Next we present results which give an idea of the reduction identified by the ICG. Finally we derive asymptotic bounds on the expected cost of the minimum feasible assignment for a class of problem instances.

3.1 COMPLEXITY OF APPROXIMATION ALGORITHMS FOR LIMITED MEMORY SCHEDULING.

Here we prove that the problem of finding an approximate solution to the limited memory scheduling problem with the cost guaranteed to be within a fixed constant of the optimum cost is as hard as the problem of finding the optimum solution. Hence, if we want an efficient algorithm we may have to be satisfied with a poorer performance in some cases. This improves the possible utility of our heuristics.

We introduce some notation and definitions (Horowitz and Sahni [1977]). The abbreviation 2-LMS denotes the 2-processor limited memory scheduling problem. Let A be
an algorithm which generates a feasible solution to every instance I of an optimization problem P. Let \( F^*(I) \) be the value of the optimum solution to instance I of P and \( F'(I) \) the value of the solution found by A.

A is an exact algorithm for P if and only if for every instance I of P, \( F^*(I) = F'(I) \).

A is an absolute approximation algorithm for P if and only if for every instance I of P, \( |F^*(I) - F'(I)| \leq k \), for some constant \( k \geq 0 \).

A is an \( \epsilon \)-approximation algorithm for P if and only if for every instance I of P,

\[
\frac{|F^*(I) - F'(I)|}{F^*(I)} \leq \epsilon,
\]

for some constant \( \epsilon \geq 0 \).

We now show that there exists a polynomial-time absolute approximation algorithm for 2-LMS if and only if there exists a polynomial-time exact algorithm for 2-LMS. Thus, since 2-LMS is NP-complete (Rao et al.[1977]) it is unlikely that we can find a polynomial-time absolute approximation algorithm for 2-LMS. Whether or not there exists an \( \epsilon \)-approximation algorithm for 2-LMS is still an open question. We use the proof techniques of Sahni[1975].

Theorem 3.1
" There exists a polynomial-time absolute approximation algorithm for 2-LMS if and only if there exists a poly-
nominal-time exact algorithm for 2-LMS."

Proof.

(i) If there exists a polynomial-time exact algorithm for 2-LMS then trivially there exists a polynomial-time absolute approximation algorithm.

(ii) The proof of the converse is by construction. Given any instance I of 2-LMS we construct an instance I'. We show that by applying an absolute approximation algorithm to I' we can derive an exact solution to I. Thus if we have a polynomial-time absolute approximation algorithm we can construct a polynomial-time exact algorithm.

Assume that there exists a polynomial-time absolute approximation algorithm A for 2-LMS. F'(I) is the value of the solution found by A for instance I and F*(I) is the value of the optimum solution. For all I,

$$|F^*(I) - F'(I)| \leq k,$$

for some constant $k \geq 0$.

Given an instance I of 2-LMS construct I' by multiplying all costs by $(k + 1)$. Memory requirements of modules are unchanged. Because the feasibility of a solution depends only on memory requirements there is a one-to-one correspondence between feasible solutions to I and I'. Further, the value of any solution to I' is $(k + 1)$ times the value of the corresponding solution to I. Therefore there is also a one-to-one correspondence between optimum solutions to I and I'.
Let $F(I)$ be the cost of any solution to $I$ and let $F(I')$ be the cost of the corresponding solution to $I'$.

Assuming integer costs in $I$, for any two solutions $a$ and $b$ we have:

$$|F_a(I) - F_b(I)| = n, \text{ for integral } n \geq 0.$$  

Therefore,  

$$|F_a(I') - F_b(I')| = (k+1) \times n$$  

(1)

Now apply algorithm A to $I'$. The value of the solution found is $F'(I')$. By definition, 

$$|F'(I') - F^*(I')| \leq k$$  

(2)

From (1) and (2) we conclude that $F'(I') = F^*(I')$. Thus A always finds an optimum solution to $I'$. This defines the corresponding optimum solution to $I$.

Multiplication of costs by $(k+1)$ increases the computation requirements of A by at most a constant factor of $O(\log k)$. Thus since by assumption A is a polynomial-time algorithm we have obtained a polynomial-time exact algorithm for 2-LMS.

Q.E.D.

The implication of Theorem 3.1 is that it is not likely that there exists a polynomial-time absolute approximation algorithm for limited memory scheduling. Thus we are justified in looking for efficient algorithms which may have a poor performance in some cases.
3.2 NODE CONDENSATION IN THE ICG.

Rao et al. [1977] pointed out that the ICG can be considerably smaller than the original processor-flow graph. Here we present statistics that give an idea of the actual simplification brought about by the ICG for two classes of randomly generated graphs. Since our statistics indicate considerable node condensation they are of general interest. They are relevant to this thesis because we use them to explain the performance of our heuristic which uses the ICG.

We first describe the method of generation of random graphs and then present and discuss our statistics.

3.2.1 GENERATION OF INPUT GRAPHS.

We use two types of graphs - constant connectivity graphs and constant degree graphs. The connectivity of a graph is the probability of an edge between any pair of nodes. The degree is the number of edges incident on a node. Average degree is equal to \((n-1)\) times connectivity for an \(n\)-node graph.

To generate a constant connectivity graph with \(n\) nodes and connectivity \(p\) \((0 \leq p \leq 1)\), we first insert edges between pairs of nodes with probability \(p\). Next we check that the graph is connected, that is, there exists at least one path between every pair of nodes. Then we randomly select two nodes to represent the processors. Finally we assign edge
costs from some random distribution. The parameters we used were \( p=0.5, n=32, 42, 52, 62 \), edge costs exponentially distributed with mean 20. The selection of parameters was arbitrary.

Constant degree graphs are intended to represent programs which exhibit the locality of intermodule reference which Stone [1978] conjectures might exist in programs written for distributed processor systems. To represent localities we generate \( q \) independent subsets of module-nodes. Each subset has \( m \) nodes and has connectivity \( p_m \). The edge weights are randomly distributed. Two processor-nodes are added and each is connected to every module-node. These edge weights also are randomly distributed. The total number of nodes in the graph,

\[
n = q \times m + 2
\]

The average degree of each module-node,

\[
d_m = p_m \times (m-1) + 2 \text{ (independent of } n)\]

The parameters that we used were \( p_m=0.8, m=10, q=3, 4, 5, 6, 7, 8 \), module communication costs exponentially distributed with mean 20, execution costs exponentially distributed with mean 25. Thus \( n=32, 42, 52, 62, 72, 82 \) and \( d_m=9.2 \). Since there do not exist any detailed analyses of programs written for distributed processors the choice of parameters is arbitrary.

The program was written in FORTRAN and run on an IBM
The maxflow, mincut algorithm we used is due to Edmonds and Karp [1972]. No attempt was made to write particularly efficient code. The execution time statistics that we present should be interpreted with these facts in mind.

Tables 3.1a and 3.1b show the minimum, average and maximum sizes of ICGs obtained for various sizes of input graphs. The size of the ICG includes the two processor-nodes. Also shown are the standard deviation of the ICG size and the mean reduction ratio (that is, the ratio of input size to ICG size). These statistics are presented for constant connectivity and constant degree graphs. In Table 3.2 are given the sample size and the mean CPU time for each case. Fig. 3.1 shows a plot of mean reduction ratio versus input size for both types of inputs.

3.2.2 DISCUSSION.

Considering first the constant connectivity graphs we find a considerable reduction in size ranging on the average from a factor of 6.3 to 20.7. Even in the worst case shown the reduction ratio is 1.8. This degree of node condensation can be explained by the relatively high degree of communication. Each module is connected to on the average n/2 other modules. There are few localities, in many cases, only one. Therefore we hypothesize that programs in which modules communicate with a large fraction of other modules may not be suitable for distribution.
### Table 3.1a Simulation Results for Constant Connectivity Graphs

<table>
<thead>
<tr>
<th>Size of Input Graphs</th>
<th>Size of Inclusive-Cuts Graph</th>
<th>Mean Reduction Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Mean</td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>3.1</td>
</tr>
<tr>
<td>42</td>
<td>3</td>
<td>5.2</td>
</tr>
<tr>
<td>52</td>
<td>3</td>
<td>5.6</td>
</tr>
<tr>
<td>62</td>
<td>3</td>
<td>3.0</td>
</tr>
</tbody>
</table>

### Table 3.1b Simulation Results for Constant Degree Graphs

<table>
<thead>
<tr>
<th>Size of Input Graphs</th>
<th>Size of Inclusive-Cuts Graph</th>
<th>Mean Reduction Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Mean</td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>6.5</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
<td>8.4</td>
</tr>
<tr>
<td>52</td>
<td>3</td>
<td>9.6</td>
</tr>
<tr>
<td>62</td>
<td>6</td>
<td>9.5</td>
</tr>
<tr>
<td>72</td>
<td>5</td>
<td>10.2</td>
</tr>
<tr>
<td>82</td>
<td>7</td>
<td>13.9</td>
</tr>
</tbody>
</table>
Table 3.2 Time for Generation of the ICG.

<table>
<thead>
<tr>
<th>Size of Input Graphs</th>
<th>Constant Connectivity Graphs</th>
<th>Constant Degree Graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sample Size</td>
<td>Mean CPU time, secs.</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>10.2</td>
</tr>
<tr>
<td>42</td>
<td>8</td>
<td>32.3</td>
</tr>
<tr>
<td>52</td>
<td>9</td>
<td>68.2</td>
</tr>
<tr>
<td>62</td>
<td>7</td>
<td>99.9</td>
</tr>
<tr>
<td>72</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>82</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Fig. 3.1 Mean Reduction Ratio for Two Classes of Graphs.
An alternate view is that the total communication cost of each module is significantly greater than its execution cost. Thus the assignments are likely to be based mainly on communication costs. We emphasize that these are only hypotheses because of the small sample sizes.

Turning to the constant degree graphs we see that the reduction ratio is more nearly a constant. This is not unexpected since condensation is restricted to nodes within the same subset due to the independence of subsets. Since subsets have the same parameters, on the average there should be the same amount of condensation in each. The average reduction of these graphs is less than half that of the constant connectivity graphs. This is due to the higher ratio of communication to execution costs in the latter case. We conjecture that programs which exhibit locality and which have a low ratio of communication cost to execution cost may be good candidates for distributed processing. If this is true then programs should be analysed to identify those classes of programs which may be suitable for distributed processing and those which need not be considered. Thus a distributed processing system may have several scheduling policies to be applied to different program categories.

In the next chapter we use these statistics to explain the better performance of a heuristic which uses the ICG when compared to a similar heuristic which does not do so.
3.3 **ASYMPTOTIC ANALYSIS OF LIMITED MEMORY SCHEDULING.**

In this section we examine the expected values of the minimum cost and minimum cost feasible solutions for large input graphs. This gives us a bound on the expected performance of the heuristics described in the next chapter.

We consider constant degree graphs as defined in section 3.2. Let \( n \) be the number of modules and \( S \) and \( T \) be the two processors. The average degree of each module is \( d + 2 \). Communication and execution costs are exponentially distributed with means \( \bar{c}_c \) and \( \bar{c}_e \) respectively. For an assignment \( I \), let \( C_I \) be the cost of the assignment. Define the normalized cost of assignment \( I \) as

\[
C_{I, \text{norm}} = \frac{C_I}{n}.
\]

Throughout this section \( n \) is assumed to be large.

For constant degree graphs we derive bounds on the expected cost of the mincut and extend these to the minimum feasible cut. To obtain a lower bound we consider only execution costs. The mincut is found by assigning each module to the processor on which it has the lower execution cost. Ties are broken arbitrarily. Refering to Fig. 3.2 let \( I \) be the mincut. Clearly, the cost of the mincut,

\[
C_I = \sum_{i=1}^{n} \min(c_{S,i}, c_{T,i}),
\]

where \( c_{i,j} \) is the weight on edge \((i,j)\).
Cut $I = \text{mincut}$.

Fig. 3.2  Processor-flow graph with only execution costs.
\{c_{S,i}\} and \{c_{T,i}\} are exponentially distributed with mean \(\overline{c}_e\). Hence \(\{\min(c_{S,i}, c_{T,i})\}\) is exponentially distributed with mean \(\overline{c}_e/2\). Hence the expected cost of the mincut,
\[\overline{C}_I = n \times \overline{c}_e/2\]  
(1)

The expected normalised cost,
\[\overline{C}_I, \text{norm} = \overline{c}_e/2\]  
(2)

Since edge costs are independent there is an equal probability of a module being assigned to either processor. Hence the expected number of modules assigned to T,
\[\overline{n}_T = n/2\]  
(3)

Now consider the case when communication costs are added. If the mincut changes from I in Fig. 3.2 then the net execution cost contribution to the mincut will increase. In addition there is the contribution from communication costs. Thus \(n\overline{c}_e/2\) is clearly a lower bound on the expected value of the mincut.

We now derive an upper bound for the expected cost of the mincut. The cost of an arbitrary cut I that assigns \(n_T\) nodes to T and \(n_S\) nodes to S consists of an execution cost component \(C_{I,e}\) and a communication cost component \(C_{I,c}\). For each module i either edge (S,i) or edge (T,i) is included in the cut. Thus the expected execution cost component of cut I,
\[C_{I,e} = n \cdot \overline{c}_e\]  
(4)

The probability of an edge between any pair of module-
nodes,

\[ p_m = \frac{d}{(n-1)} \]

Thus each module assigned to T communicates with on the average \( p_m \cdot n_S \) modules assigned to S. This contributes \( p_m \cdot n_S \cdot \bar{c}_c \) to the expected communication cost. Summing over all \( n_T \) modules assigned to T,

\[ \bar{c}_{I,c} = p_m \cdot n_S \cdot \bar{c}_c \cdot n_T \]

Since \( n_S + n_T = n \), we have the expected communication cost as a function of \( n_T \),

\[ \bar{c}_{I,c}(n_T) = \frac{d \cdot \bar{c}_c \cdot n_T \cdot (n-n_T)}{(n-1)} \]  

(5)

Thus the expected cost of an arbitrary cut as a function of \( n_T \),

\[ \bar{c}(n_T) = \frac{d \cdot \bar{c}_c \cdot n_T \cdot (n-n_T)}{(n-1)} + n \cdot \bar{c}_e \]  

(6)

Since \( n \approx (n-1) \) for large \( n \), we have the normalized cost,

\[ \bar{c}_{\text{norm}}(n_T) = \frac{d \cdot \bar{c}_c \cdot n_T \cdot (1 - n_T)}{n} + \bar{c}_e \]  

(6a)

A plot of \( \bar{c}_{\text{norm}}(n_T) \) against \( n_T/n \) is shown in Fig. 3.3. It is seen to be an inverted parabola with minima of \( \bar{c}_e \) at \( n_T/n = 0 \) and \( 1 \) and a maximum of \( \frac{d \cdot \bar{c}_c}{4} + \bar{c}_e \) at \( n_T/n = 0.5 \).

In every instance the value of the mincut is less than or equal to the value of the cut that assigns all the nodes to one processor (i.e. \( n_T/n = 0, 1 \)). Thus the expected normalized value of the mincut is no greater than \( \bar{c}_{\text{norm}}(n) = \bar{c}_e \) (from (6a)). Thus \( n \cdot \bar{c}_e \) is an upper bound of the expected cost of the mincut.
Fig. 3.3 Expected Normalized Cost of an Arbitrary Cut.
For certain cases we can obtain a tighter bound. The expected cost of cut I in Fig. 3.2 when communication costs are considered becomes,

\[ \bar{C}_I = n \cdot \bar{c}_e / 2 + \bar{c}_I, c (n/2) \]

\[ = n \cdot \bar{c}_e / 2 + d \cdot \bar{c}_c \cdot n / 4, \text{ from (1) and (5).} \]

Defining \( r = \frac{d \cdot \bar{c}_c}{\bar{c}_e} \), the ratio of communication cost to execution cost, we have,

\[ \bar{c}_I = \frac{n \cdot \bar{c}_e}{2} \cdot (1 + \frac{d \cdot \bar{c}_c}{2 \cdot \bar{c}_e}) \]

or

\[ \bar{c}_I = \frac{n \cdot \bar{c}_e}{2} \cdot (1 + r/2) \quad (7) \]

Clearly in every instance the value of the mincut must be less than or equal to that of cut I. Hence the expected cost of the mincut is bounded from above by

\[ \min(\frac{n \cdot \bar{c}_e}{2} \cdot (1 + r/2), \ n \cdot \bar{c}_e) \]

For \( r < 2 \) the first term is less than the second term.

Thus for the expected cost of the mincut we have the inequality,

\[ n \cdot \bar{c}_e / 2 \leq \bar{c}_{\text{min}} \leq \min(\frac{n \cdot \bar{c}_e}{2} \cdot (1 + r/2), \ n \cdot \bar{c}_e) \quad (8) \]

Assume that one processor, say T, has a limited memory. We obtain bounds on the expected cost of the minimum feasible cut \( \bar{c}_{\text{mfc}} \). Clearly, \( \bar{c}_{\text{mfc}} \geq \bar{c}_{\text{min}} \).
Thus \( \bar{c}_{mfc} \geq n \cdot \bar{c}_e / 2 \) \hspace{1cm} (9a)

However, since cut I in Fig. 3.2 may not be feasible, this cut is not an upper bound on the minimum feasible cut. Clearly it is feasible to assign all nodes to S. Thus we have

\[ \bar{c}_{mfc} \leq n \cdot \bar{c}_e \] \hspace{1cm} (9b)

Under certain conditions we can tighten the bound in (9b). We do this by considering the expected cost of the solution found by a heuristic which we describe below.

3.3.1 **HEURISTIC RANDOM.**

For the constant-degree model with parameters given above, let processor T have a memory capacity of \( M_T \). Denote the mean of the module memory requirements by \( \bar{m} \) and the total program size by \( M \).

We describe the algorithm RANDOM. Initially RANDOM assigns modules at random to S until the unassigned modules can be accommodated in the memory of T. Let R be the cut thus found (Fig. 3.4). Let the set of modules assigned to S be \( N_{S,R} \) and the cardinality of this set be \( n_0 = |N_{S,R}| \).

Now for \( i = 0, 1, 2, \ldots, (n-n_0) \), RANDOM iterates as follows:-

Randomly select \( i \) of the unassigned modules and assign them to S which now has \( n^i_S = (n_0 + i) \) modules. For the remaining \( n_1 = (n-n^i_S) \) modules, considering only execu-
Fig. 3.4 Cuts Found by RANDOM.
tion costs, find the mincut. Assign the \( n_1 \) nodes to \( S \) and \( T \) according to this mincut. We have thus found a feasible assignment (cut \( I_i \) in Fig. 3.4). RANDOM now increments \( i \) by 1 and repeats this step, initially assigning to \( S \) the same set \( N_{S,R} \).

The solution found by RANDOM is the cut \( I_i \) with the lowest cost among all the cuts \( I_i, 0 \leq i \leq n_0 \).

RANDOM performs at most \( n \) iterations. Each iteration requires \( O(n) \) additions and comparisons. Thus in the worst case RANDOM takes \( O(n^2) \) time and is therefore very efficient.

We now derive an expression for the expected cost of the solution found by RANDOM.

The expected value of \( n_0 \),

\[
\bar{n}_0 = \frac{(M - M_T)}{\bar{m}} \tag{10}
\]

In any iteration,

\[
n_S^i = n_0 + i \tag{11}
\]

Hence the size of the subset for which RANDOM finds a mincut,

\[
n_1 = n - n_S^i = n - (n_0 + i)
\]

For a given \( i \), the expected value of \( n_1 \),

\[
\bar{n}_1 = n - (\bar{n}_0 + i)
\]

From (3) the expected number of nodes assigned to \( T \),

\[
\bar{n}_T = \frac{n_1}{2} = \frac{n}{2} - \left( \bar{n}_0 + \frac{i}{2} \right) \tag{12}
\]
From (11) we have the expected value of $n'_S$

$$\overline{n}'_S = \overline{n}_0 + i$$  \hspace{1cm} (13)

The cost of cut $I_i$ consists of three components: the total communication cost, the execution cost of the $n'_S$ nodes randomly assigned to $S$ and the execution cost for the $n_1$ nodes for which a mincut was found.

From (5) the expected value of the first component is,

$$C_{I_i,C}(\overline{n}_T) = \frac{d \cdot \overline{c}_c \cdot \overline{n}_T \cdot (n - \overline{n}_T)}{(n-1)}$$

The second component is clearly $\overline{n}'_S \cdot \overline{c}_e$.

From (1) the third component is $\overline{n}_T \cdot \overline{c}_e / 2$.

Thus the expected cost of cut $I_i$,

$$\overline{C}_{I_i} = \frac{d \cdot \overline{c}_c \cdot \overline{n}_T \cdot (n - \overline{n}_T)}{(n-1)} + \overline{n}'_S \cdot \overline{c}_e + \overline{n}_T \cdot \overline{c}_e / 2$$

Substituting the expressions for $\overline{n}_T$ and $\overline{n}'_S$ from (12) and (13) and since $n_0 = n-1$, we have,

$$\overline{C}_{I_i} = \frac{d \cdot \overline{c}_c \cdot (n^2 - (\overline{n}_0 + i)^2)}{4n} + \frac{\overline{c}_e}{2} \cdot (n + \overline{n}_0 + i)$$

Replacing $(\overline{n}_0 + i)/n$ by the variable $\overline{x}$,

$$\overline{C}_I(\overline{x}) = \frac{n \cdot \overline{c}_e}{2} \cdot \left[ (1 + \overline{x}) + \frac{r}{2} \cdot (1 - \overline{x}^2) \right] \hspace{1cm} (14)$$

where $r = d \cdot \overline{c}_c / \overline{c}_e$

Normalizing,

$$\overline{C}_{I,norm}(\overline{x}) = \frac{\overline{c}_e}{2} \cdot \left[ (1 + \overline{x}) + \frac{r}{2} \cdot (1 - \overline{x}^2) \right] \hspace{1cm} (14a)$$
Fig. 3.5 shows the variation of $\overline{C}_{I,norm}(\overline{x})$ with $\overline{x}$ for various values of $r$. For each value of $r$ we plot two curves, one obtained from (14a) and the other obtained by simulation. For each value of $r$ RANDOM was tested on 100 randomly generated constant degree graphs each of size 102 nodes. The analytical and simulation results agree closely.

For any memory capacity $M_T$ on $T$ we have the expected fraction of the program that cannot be assigned to $T$,

$$\overline{F}_0 = \overline{n}_0/n = (M - M_T)/M \quad \text{(from (10))}.$$  

The expected costs of the cuts $I_i$ found by RANDOM lie in the interval $[\overline{F}_0 \leq \overline{x} \leq 1]$. The expected cost of the solution found by RANDOM is clearly the minimum of $\overline{C}_{I}(\overline{x})$ in the above interval. For $r \geq 2$ this minimum is $\overline{n} \cdot \overline{c}_e$ (Fig. 3.5) and for $r < 2$ it is $\overline{C}_{I}(\overline{F}_0)$.

Since the solution of RANDOM is feasible we now have an upper bound on the expected value of the minimum feasible cut,

$$\overline{C}_{mfc} \leq \min \left( \overline{C}_{I}(\overline{F}_0), \overline{n} \cdot \overline{c}_e \right)$$

or,

$$\overline{C}_{mfc} \leq \min \left( \frac{n \cdot \overline{c}_e}{2} \cdot [1 + \overline{F}_0] + \frac{r}{2} \cdot (1 - \overline{F}_0)^2], \overline{n} \cdot \overline{c}_e \right) \quad (15)$$

Together with the lower bound in (9a) this yields the inequality,

$$\overline{n} \cdot \overline{c}_e/2 \leq \overline{C}_{mfc} \leq \min \left( \frac{n \cdot \overline{c}_e}{2} \cdot [1 + \overline{F}_0] + \frac{r}{2} \cdot (1 - \overline{F}_0)^2], \overline{n} \cdot \overline{c}_e \right) \quad (15a)$$
Fig. 3.5 Expected value of cuts found by RANDOM.
First we consider the expected error in the solution found by RANDOM. Define \( \epsilon = \frac{|C_{\text{heuristic}} - C_{\text{optimum}}|}{C_{\text{optimum}}} \).

From (15a) we have a bound on the expected error of RANDOM,

\[
\epsilon \leq \min \left\{ \frac{n \cdot \bar{c}_e}{2} \left[ (1 + \bar{f}_0) + \frac{r}{2} \cdot (1 - \bar{f}_0^2) \right], \frac{n \cdot \bar{c}_e}{2} \right\} - \frac{n \cdot \bar{c}_e}{2}
\]

or,

\[
\epsilon \leq \min \left\{ \left[ \bar{f}_0 + \frac{r}{2} \cdot (1 - \bar{f}_0^2) \right], 1 \right\}
\]

Thus for \( r \geq 2 \), the expected error in the solution found by RANDOM is no more than 100%. For \( r < 2 \), that is if communication costs are not large compared to execution costs, the expected error may be strictly less than 100%.

Since RANDOM is very efficient we can always use it in addition to any other heuristic and take as the solution of the composite heuristic the lower of the solutions found independently by RANDOM and the other heuristic of interest. Thus the bound on the expected performance of RANDOM is also a bound for any such composite heuristic.

Consider any heuristic which examines several feasible cuts and chooses as its solution the cut with lowest cost. Assume that one of the cuts examined is always the cut assigning all nodes to S. This cut clearly has expected
value \( n \cdot \overline{c}_e \) since it contains only the edges \((i,T)\) with cost \( c_{i,T}, 1 \leq i \leq n \). Hence any such heuristic has an expected error,
\[
\bar{\varepsilon} \leq \frac{|n \cdot \overline{c}_e - n \cdot \overline{c}_e/2|}{n \cdot \overline{c}_e/2}
\]
or,
\[
\bar{\varepsilon} \leq 1
\]
Thus for constant degree graphs any such heuristic on the average for large \( n \) finds a solution that costs no more than twice the optimum solution. This is a characteristic of the two heuristics discussed in the next chapter.

We note that the results of this section are valid even if there is a particular subset of modules, independent of \( n \), which must be assigned to a specific processor due to special capabilities provided only by that processor. This can be seen easily by considering normalized costs of assignments. The maximum cost that these modules together can contribute to any cut is constant. Thus, when normalized, for large \( n \) this is vanishingly small.

3.4 **SUMMARY**.

In this chapter we have proved that an absolute approximation algorithm for the 2-processor limited memory scheduling problem is no more efficient than an exact algorithm. Thus unless \( P = NP \) we cannot efficiently find an absolute approximation solution in every instance of the problem. We then presented statistics on the reduction
in the ICG for constant connectivity and constant degree graphs. Finally we derived asymptotic bounds on the expected value of the minimum feasible cut for constant degree graphs.

These results motivate our development of heuristics and are used in the discussion of their performance in the next chapter.
We are now ready to present our development of heuristics for 2-processor scheduling with limited memory. If we wish to guarantee an exact or absolute approximation solution in every instance we may have to accept an exponential-time complexity in some cases. Our heuristics have polynomial-time complexity in every instance but may sometimes have a poor performance.

The results of section 3.2 indicate that use of the ICG may appreciably reduce the problem size. This motivates the development of a heuristic based on the ICG. We also present a similar heuristic which uses only the processor-flow graph. This serves as a control to indicate the utility of the ICG. It also provides some measure of confirmation for the results of section 3.3.

We first describe the two heuristics informally and formally and derive their worst-case time complexities. Then we present results obtained for constant degree graphs. This is followed by an intuitive explanation of the observed performance and a discussion of the implications.

4.1 *A Heuristic Based on the ICG.*

The heuristic described here is a "greedy" algorithm
(Horowitz and Sahni [1977]) which is applied to the ICG of a processor-flow graph. This procedure selects one node at a time to assign to $T$, the processor with limited memory. This step is repeated until the procedure cannot assign any more nodes to $T$. The selections are made in accordance with the partial ordering imposed by the ICG. After each selection we evaluate the cost of the assignment. The solution found by the heuristic is the lowest valued of the assignments thus obtained.

At each iteration we select the node with the largest saving in cost per unit of memory. Informally, we consider memory to consist of a limited number of units. At each step we "buy" the object which gives the most return per unit "spent". The saving per unit of memory is the index of the node.

We make the selections in accordance with the partial ordering of the ICG by considering at each iteration only those nodes adjacent to nodes already selected. Initially we consider nodes adjacent to $T$. Further, we do not consider a node $N$ until all nodes on every path from $T$ to $N$ have been selected.

The saving in selecting a node is the net reduction in

1 Node $i$ is adjacent to node $j$ if and only if there is an edge in the ICG directed from $i$ to $j$. 
communication and execution costs due to reassigning the node from S to T. This saving may be negative. It can be computed in $O(m)$ time, where $m$ is the number of nodes in the ICG. However the change in communication cost depends on the set of nodes already assigned to T. Thus, at every iteration the saving would have to be recomputed for every node under consideration, increasing the amount of computation required. This is compounded if we are interested in assignments for a range of memory capacities on T.

For efficiency we select a measure of the saving associated with selecting a node that is invariant with respect to node assignments. Thus we avoid having to recompute the savings for every node in each iteration. Refering to the ICG in Fig. 4.1 assume that at some stage in the application of the algorithm we have reached the cut I. Consider the selection of node $P_3$. The saving is the cost of cut I minus the cost of cut II. Intuitively, if nodes are in different partial orders they are in separate localities and therefore the communication cost between them is probably not appreciable. Thus the difference in cost between cuts III and IV is a reasonable approximation to the difference in cost between cuts I and II. But cut III is $P_3(S)$, the mincut reassigning $P_3$ to S and likewise cut IV is $P_2(S)$. These have been determined during the construction of the ICG. Hence we use this
Fig. 4.1 An ICG.

Fig. 4.2 An ICG.
difference as the desired measure of the saving in selecting \( P_3 \) as it is invariant with respect to node assignments.

Now consider the ICG shown in Fig. 4.2. Assume that we have reached the intermediate cut I and are considering the selection of \( P_3 \). We cannot make the same approximation as in the previous case since we do not have the value of any cut assigning \( P_1 \) and \( P_2 \) to S and \( P_3 \) to T. We do however have the values of cuts \( P_1(S) \) and \( P_2(S) \). \( P_1(S) \) and \( P_2(S) \) are mincuts reassigning \( P_1 \) and \( P_2 \) respectively to S. Thus any cut that reassigns both \( P_1 \) and \( P_2 \) to S cannot have a value less than the larger of the values of \( P_1(S) \) and \( P_2(S) \). Hence as an approximation to the value of cut II we choose the larger of the values of \( P_1(S) \) and \( P_2(S) \).

\[
\text{cost of } P_1(S) - \max(\text{cost of } P_j(S))
\]

In general, index\((P_i)\) = \[
\frac{\text{cost of } P_i(S) - \max(\text{cost of } P_j(S))}{\text{Size}(P_i)}
\]

Clearly this index is independent of node assignments and the memory capacity of T.

We now state the heuristic formally.

**Algorithm INDEX-I.**

**Input.**

1. An n-node flow graph in the form of an n x n array \( \text{COST} \), where \( \text{COST}(i,j) \) is the weight on edge \( (i,j) \)
(2) For each module $M$ the size of the module, $\text{SIZE}(M)$
(3) The memory capacity of processor $T$, $\text{MEM}(T)$.

**Output.**
A possibly sub-optimal feasible assignment of modules and the cost of the assignment.

**Method.**
(1) (Find the ICG.)
Construct the ICG of the given flow graph using the algorithm described by Rao et al. [1977]. Hereafter we refer to the condensed modules of the ICG.

(2) (Calculate indices.)
For each node $M$ find $X_M$, the set of successor nodes of $M$. ($J$ is a successor of $M$ if and only if $(M,J)$ is an edge in the ICG.) Calculate $\text{INDEX}(M) = \text{PROFIT}(M)/\text{SIZE}(M)$, where

$$\text{PROFIT}(M) = C(M) - \max_{i \in X_M} C(i),$$

where $C(i)$ is the value of the cut reassigning $i$ to $S$.

(3) (Initialization.)
Insert into the list of candidate nodes all and only the successors of $T$.
$\text{T-NODES} \leftarrow \{T\}$, where $\text{T-NODES}$ is the set of nodes assigned to $T$ in the solution.
$\text{INT} \leftarrow \{T\}$, where $\text{INT}$ is the set of nodes assigned to $T$ at some intermediate stage.
$\text{VALMIN} \leftarrow \sum_{i=1}^{n} \text{COST}(i,T)$, where $\text{VALMIN}$ is the cost of the solution.
VALUE + VALMIN, where VALUE is the cost of some intermediate assignment.

MEMREQ + 0, where MEMREQ is the total size of the nodes assigned to T in INT.

(4) (Selection of nodes.)

Iterate until the list of candidate nodes is empty:

Select the candidate node \( N \) with the largest index.
Delete \( N \) from the list of candidate nodes.

If \( \text{MEMREQ} + \text{SIZE}(N) \leq \text{MEM}(T) \) then:

\[
\text{INT} \leftarrow \text{INT} + \{N\}
\]
\[
\text{MEMREQ} \leftarrow \text{MEMREQ} + \text{SIZE}(N)
\]

Add to the list of candidates all nodes \( P \) such that (i) \( P \) is a successor of \( N \), (ii) \( P \neq S \) and (iii) every predecessor of \( P \) is in \( \text{INT} \) (\( Q \) is a predecessor of \( P \) if and only if \( (Q,P) \) is an edge in the ICG).

\[
\text{VALUE} \leftarrow \text{VALUE} + \sum_{i \notin \text{INT}} \text{COST}(i,N) - \sum_{i \in \text{INT}} \text{COST}(i,N).
\]

If \( \text{VALUE} < \text{VALMIN} \) then: (found a new minimum)

\[
\text{VALMIN} \leftarrow \text{VALUE}
\]
\[
\text{T-NODES} \leftarrow \text{INT}
\]

endif.

endif.

(5) (Output.)

Assign to \( T \) the nodes in \( \text{T-NODES} \) and the remainder to \( S \).
The cost of the assignment is \( \text{VALMIN} \).
We illustrate the heuristic using the graph shown in Fig. 4.3a. Module sizes are given in Table 4.1a. The ICG constructed from the graph is given in Fig. 4.3b. Table 4.1b indicates the calculation of indices. Note that the memory requirement of D' is the sum of the requirements of D, E and F since D' is formed by condensing D, E and F.

Initially the list of candidates consists of G. We assign G to T, delete it from the list and insert D'. D' is also deleted from the list, assigned to T and C and B are inserted. MEMREQ is now 21. We select the node with larger index, B. MEMREQ+SIZE(B)=23 < MEM(T) so we assign B to T and delete it from the list of candidates. Next we consider C. MEMREQ+SIZE(C)=25 > MEM(T) so C is deleted. The list is empty and the algorithm terminates. Table 4.1c shows the assignments found in each iteration with the costs of the assignments. The assignment found in iteration #3 has the lowest cost, 28. This is the solution found by the heuristic. In this case it is also the minimum cost feasible assignment.

We now establish the worst-case time complexity of the heuristic algorithm.

**Theorem 4.1.**

"Algorithm INDEX-I has a worst-case time complexity of $O(n^2e)$, where $n$ is the number of nodes in the input graph.
Fig. 4.3a Processor-flow Graph.

Min-cut
Cost = 19

S = \{S, A\}
D' = \{D, E, F\}

Fig. 4.3b Inclusive-cuts Graph.
Table 4.1a  Module Sizes.

<table>
<thead>
<tr>
<th>Module, M</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE(M)</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Memory Capacity of T, MEM(T) = 24

Table 4.1b  Indices of Modules.

<table>
<thead>
<tr>
<th>Module, M</th>
<th>PROFIT(M)</th>
<th>SIZE(M)</th>
<th>INDEX(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>16</td>
<td>2</td>
<td>8.0</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>D'</td>
<td>2</td>
<td>11</td>
<td>0.2</td>
</tr>
<tr>
<td>G</td>
<td>6</td>
<td>10</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 4.1c  Intermediate Assignments.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Modules Assigned to S</th>
<th>Modules Assigned to T</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B,C,D',G</td>
<td>-</td>
<td>43</td>
</tr>
<tr>
<td>1</td>
<td>B,C,D'</td>
<td>G</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>B,C</td>
<td>D',G</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>B,D',G</td>
<td>28</td>
</tr>
</tbody>
</table>
and $e$ the number of edges."

Proof.

Construction of the ICG in step 1 takes $O(n^2e)$. This is due to $O(n)$ executions of the maxflow, mincut algorithm each of which takes $O(ne)$ (Rao et al.[1977], Karzanov [1974]).

Step 2 takes $O(n^2)$.

The operations performed on the list of candidate nodes in steps 3 and 4 are INSERT, DELETE and MAX. (Aho, Hopcroft, Ullman [1975]). Each of these operations is performed at most $O(n)$ times. This can be done in $O(n \cdot \log n)$ time using a heap, 2-3 tree or AVL tree to represent the list. Use of a linked list representation results in a time of $O(n^2)$.

Step 5 is of complexity $O(n)$.

Thus the worst-case time complexity of the algorithm INDEX-I is $O(n^2e)$.

Q.E.D.

Algorithm INDEX-I is not an $\epsilon$-approximation algorithm for the 2-processor limited memory scheduling problem. This is proved by the example in Fig. 4.4. For $x \geq 0$, the optimum feasible assignment assigns A to S and B to T and has a cost of $C^* = 20$. Applying heuristic INDEX-I we have:
Fig. 4.4 Proof that INDEX-I is not $\varepsilon$-approximate.
INDEX(A) = (20 - 10)/10 = 1
INDEX(B) = (x + 30 - 10)/SIZE(B)
= (x + 20)/(x + 25) < 1, for x ≥ 0.

Thus for x ≥ 0 INDEX-I assigns A to T and B to S. This assignment has a cost C' = x + 30.

Therefore, ε = |C* - C'|/C* = (x + 10)/20

We can make ε arbitrarily large by suitably choosing x.

Apers [1978] shows that the average complexity of the Dinic-Karzanov algorithm is O(n^2) for certain classes of random graphs. Thus for these classes of inputs INDEX-I has an expected complexity of O(n^3).

4.2 A SECOND HEURISTIC.

We now describe a heuristic based on the same principles as INDEX-I but which does not involve construction of the ICG. Assume that we have found a minimum cost assignment using a maxflow, mincut algorithm and that the assignment is not feasible. We condense into S the nodes assigned to S and now consider the reduced graph. We iterate assigning a node to T based on an index defined below.

Assume that we have found an intermediate assignment. The index of a node assigned to S is the ratio of the saving obtained by reassigning the node to T to the memory requirement of the node. The saving may be negative. The heuristic now proceeds exactly as the heuristic INDEX-I
except that there is no partial ordering on the nodes and the indices may change with each reassignment.

We now present the algorithm formally.

**Algorithm INDEX-G.**

**Input.**
(1) An n-node flow graph in the form of an n x n array COST, where COST(i,j) is the weight on edge (i,j).
(2) For each module M, SIZE(M), the size of the module.
(3) The memory capacity of processor T, MEM(T).

**Output.**
A possibly sub-optimal feasible assignment of modules and the cost of the assignment.

**Method.**
(1) (Find the mincut.)
Find a mincut between S and T using a maxflow, mincut algorithm (Dinic [1970], Karzanov[1974]). Condense into S the nodes assigned to S. In the remainder of the algorithm we consider only nodes assigned to T by the mincut.

(2) (Initialization.)
MEMREQ ← 0, where MEMREQ is the total size of nodes assigned to T.
T-NODES ← {T}, where T-NODES is the set of nodes
assigned to $T$ by the lowest valued feasible cut encountered.

$\text{INT} \leftarrow \{T\}$, where $\text{INT}$ is the set of nodes assigned to $T$ by an intermediate cut.

$\text{VALMIN} \leftarrow \sum_{i=1}^{n} \text{COST}(i,T)$, where $\text{VALMIN}$ is the cost of the lowest valued cut encountered.

$\text{VALUE} \leftarrow \text{VALMIN}$, where $\text{VALUE}$ is the value of an intermediate cut.

For all nodes $i$ compute initial indices:

$\text{PROFIT}(i) \leftarrow \sum_{j \notin \text{INT}} \text{COST}(i,j) - \sum_{j \in \text{INT}} \text{COST}(i,j)$

$\text{INDEX}(i) \leftarrow \text{PROFIT}(i)/\text{SIZE}(i)$

Initialize the list of candidate nodes to include all nodes except $S$ and $T$.

(3) (Selection of nodes for assignment to $T$.)

Iterate until the list of candidates is empty:

Find the candidate node $M$ with maximum $\text{INDEX}$. Delete $M$ from the list of candidates.

If $\text{MEMREQ} + \text{SIZE}(M) \leq \text{MEM}(T)$ then:

$\text{INT} \leftarrow \text{INT} + \{M\}$

$\text{VALUE} \leftarrow \text{VALUE} - \text{PROFIT}(M)$

$\text{MEMREQ} \leftarrow \text{MEMREQ} + \text{SIZE}(M)$

If $\text{VALUE} < \text{VALMIN}$ then: (found a new minimum)

$\text{VALMIN} \leftarrow \text{VALUE}$

$\text{T-NODES} \leftarrow \text{INT}$

endif.
For all candidate nodes: (recompute indices)

\[ \text{PROFIT}(i) + \text{PROFIT}(i) + 2 \cdot \text{COST}(i,M) \]

\[ \text{INDEX}(i) = \frac{\text{PROFIT}(i)}{\text{SIZE}(i)}. \]

endif.

(5) (Output.)

Assign to T the nodes in T-NODES and the remainder to S. The cost of the assignment is VALMIN.

***

As an illustration we apply INDEX-G to the flow graph shown in Fig. 4.3a. Table 4.2a shows the calculation of indices in each iteration. The index of the selected node is circled. Table 4.2b shows the intermediate cuts obtained. The cut found in iteration #1 has the minimum value of 37. This is the solution found by INDEX-G. We note that the minimum feasible solution has a cost of 28.

Theorem 4.2 establishes the worst-case time complexity of heuristic INDEX-G.

Theorem 4.2.

"Algorithm INDEX-G has a worst-case time complexity of \(O(n \cdot \max(n, e))\), where \(n\) is the number of nodes in the input graph and \(e\) the number of edges."

Proof.

Finding the mincut in step 1 takes \(O(ne)\) (Dinic [1970], Karzanov [1974], Even [1976]).
### Table 4.2a Index Calculation in INDEX-G.

<table>
<thead>
<tr>
<th>Module</th>
<th>Module Size</th>
<th>Iteration # 1</th>
<th>Iteration # 2</th>
<th>Iteration # 3</th>
<th>Iteration # 4</th>
<th>Iteration # 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Profit</td>
<td>Index</td>
<td>Profit</td>
<td>Index</td>
<td>Profit</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>-18</td>
<td>-9</td>
<td>-18</td>
<td>-9</td>
<td>-3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>-11</td>
<td>-5.5</td>
<td>-11</td>
<td>-5.5</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>-35</td>
<td>-17.5</td>
<td>-35</td>
<td>-17.5</td>
<td>-25</td>
</tr>
<tr>
<td>E</td>
<td>7</td>
<td>-36</td>
<td>-5.1</td>
<td>-12</td>
<td>-1.7</td>
<td>-</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>-10</td>
<td>-5</td>
<td>-10</td>
<td>-5</td>
<td>-10</td>
</tr>
<tr>
<td>G</td>
<td>10</td>
<td>6</td>
<td>0.6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
## Table 4.2b Intermediate Cuts in INDEX-G.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Modules Assigned to T</th>
<th>Cost</th>
<th>Size of Modules Assigned to T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>37</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>E,G</td>
<td>48</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>C,E,G</td>
<td>39</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>B,C,E,G</td>
<td>42</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>B,C,D,E,G</td>
<td>47</td>
<td>23</td>
</tr>
</tbody>
</table>
Step 2 takes $O(n^2)$ time.

There are at most $n$ iterations in step 3. In each iteration saving the assignment and recomputing indices can take $O(n)$ time. Thus step 3 takes time $O(n^2)$ in the worst case.

Step 4 takes $O(n)$ time.

Thus the worst-case time complexity of INDEX-G is $O(n \cdot \max(n, e))$.

Q.E.D.

We note that in step 1 the modules assigned to $S$ by the mincut are condensed into $S$. This may reduce the work in the succeeding steps. Applying the results due to Apers[1978] stated earlier the expected time complexity of INDEX-G is $O(n^2)$ for certain classes of input graphs.

INDEX-G is not an $\varepsilon$-approximation algorithm as can be seen by application to the processor-flow graph in Fig. 4.5.

Initial indices are:

\[
\begin{align*}
\text{INDEX}(A) &= \frac{[(x + 15 + 15) - (x + 15 + 5)]}{\text{SIZE}(A)} \\
&= 1 \\
\text{INDEX}(B) &= \frac{[(x + 15 + 15) - (15 + 5)]/\text{SIZE}(B)}{(x + 10)/(x + 25) < 1, \text{ for } x \geq 0.}
\end{align*}
\]
Fig. 4.5 Proof that INDEX-G is not $\epsilon$-approximate.
Node A is assigned to T. Node B cannot now be assigned to T because of the memory constraint. Thus we have:

Cost of solution found by INDEX-G, \( C' = x + 20 \)

Cost of minimum feasible solution, \( C^* = 20 \)

Therefore,

\[
\epsilon = \frac{|C' - C^*|}{C^*} = \frac{x}{20}
\]

By appropriately choosing \( x \) we can make \( \epsilon \) arbitrarily large.

4.3 RESULTS OF SIMULATION.

We now describe the results obtained by applying the two heuristics to several constant connectivity and constant degree graphs. The graphs were generated as described in section 3.2. Module sizes were exponentially distributed with mean 20. The memory capacity of T was set to half the total size of the modules initially assigned to T by the mincut. For reasons explained earlier these choices were arbitrary.

For constant connectivity graphs in almost every instance the minimum feasible cut assigned all the modules to S. Thus, trivially, both heuristics found the minimum feasible cut almost always.

The results obtained for constant degree graphs are summarized in Tables 4.3a, 4.3b and 4.3c. For each
<table>
<thead>
<tr>
<th>Size of Input</th>
<th>$\varepsilon_I = \frac{C_{INDEX-I} - C_{opt}}{C_{opt}} \times 100%$</th>
<th>$\varepsilon_G = \frac{C_{INDEX-G} - C_{opt}}{C_{opt}} \times 100%$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Mean</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>52</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>62</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>72</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>82</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 4.3b Percentage Error of INDEX-I and INDEX-G.

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Sample Size</th>
<th>INDEX-I</th>
<th>INDEX-G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Percentage error, $\varepsilon_I$</td>
<td>Percentage error, $\varepsilon_G$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mean</td>
<td>Standard Deviation</td>
<td>Mean</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>52</td>
<td>10</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>62</td>
<td>10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>72</td>
<td>10</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>82</td>
<td>6</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Table 4.3c Results for INDEX-I and INDEX-G.

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Sample Size</th>
<th>INDEX-I % of Optimum Solutions</th>
<th>Mean CPU time, sec.</th>
<th>INDEX-G % of Optimum Solutions</th>
<th>Mean CPU time, sec.</th>
<th>Ratio of CPU times INDEX-I/INDEX-G</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>10</td>
<td>90</td>
<td>3</td>
<td>60</td>
<td>0.6</td>
<td>5</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>80</td>
<td>5</td>
<td>50</td>
<td>1.2</td>
<td>4</td>
</tr>
<tr>
<td>52</td>
<td>10</td>
<td>70</td>
<td>20</td>
<td>10</td>
<td>1.7</td>
<td>12</td>
</tr>
<tr>
<td>62</td>
<td>10</td>
<td>90</td>
<td>40</td>
<td>10</td>
<td>3.3</td>
<td>12</td>
</tr>
<tr>
<td>72</td>
<td>10</td>
<td>70</td>
<td>44</td>
<td>0</td>
<td>3.9</td>
<td>11</td>
</tr>
<tr>
<td>82</td>
<td>6</td>
<td>50</td>
<td>61</td>
<td>0</td>
<td>5.6</td>
<td>11</td>
</tr>
</tbody>
</table>
heuristic and for each size of input graph the following are presented - the minimum, mean, maximum and standard deviation of the error in the solution expressed as a percentage of the cost of the optimum solution; the sample size; the percentage of instances in which the optimum solution was obtained; and the mean CPU time for computing a solution. The algorithms were written in FORTRAN and run on an IBM 370/155. The maxflow, mincut algorithm used is due to Edmonds and Karp [1972]. This has a worst-case complexity of $O(ne^2)$ compared to $O(ne)$ for the Dinic-Karzanov algorithm. Thus the execution times may be reducible. It is valid, however, to use them to compare the two heuristics since executions of the maxflow, mincut algorithm form a substantial part of both.

Figs. 4.6a and 4.6b show plots of mean error and percentage of optimum solutions found versus input size for both heuristics. Fig. 4.6c shows the ratio of CPU times.

From the data we note that INDEX-I finds the optimum solution in 77% of the instances used while INDEX-G does so in only 23%. Further, INDEX-I has a mean error of 2% while that of INDEX-G is 11%. However, INDEX-G is 5 to 12 times faster than INDEX-I.

We now discuss the performance of the heuristics. In an n-node graph the number of possible assignments is $2^n$. The heuristics examine at most n assignments. The fraction
Fig. 4.6a Mean Percentage Error versus n.
Fig. 4.6b  Percentage of optimum solutions versus n.
Fig. 4.6c Ratio of CPU times versus n.
of assignments examined, \( \frac{n}{2^n} \), is extremely small for the graphs considered (\( n \geq 32 \)). Thus we might expect the mean error for the heuristics to be close to the asymptotic bound of 100% (\( (3.15a) \) with \( r = 5.76 \)). However, the observed mean errors are 2% and 11% for INDEX-I and INDEX-G respectively. This could be due to one or more of the following: (i) the actual expected value of the minimum feasible cut is close to the upper limit of the interval \( \left( \frac{n \cdot C_e}{2}, n \cdot C_e \right) \) (from \( (3.15a) \)), (ii) \( n = 32, \ldots, 82 \) is not sufficiently large and (iii) the use of the ICG and the indices as defined in the two heuristics are indeed useful. We present arguments in favour of the last of these postulates.

Considering first INDEX-I, we note that on the average the ICG is smaller than the input graph by a factor of 6 (Fig. 3.1). This greatly reduces the number of assignments to be considered since the optimum assignment is in the ICG (Rao et al. [1977]). In addition, the partial ordering imposed by the ICG may further reduce the number of assignments to be considered.

By way of illustration, consider the processor-flow graph of Fig. 2.4a and the corresponding ICG in Fig. 2.4d. The number of possible assignments in the processor-flow graph is \( 2^7 = 128 \). The ICG has 4 module-nodes. Thus, ignoring the partial ordering, there are \( 2^4 = 16 \) assignments to be considered. If we consider the partial
ordering we need examine only 6 assignments to find the optimum assignment.

In addition, many of the assignments eliminated by the ICG are ones which partition localities. Such assignments have a high communication cost component and hence a high total cost. To summarize, use of the ICG can greatly reduce the number of assignments to be considered and the assignments that are considered may have an average cost lower than the average cost of the assignments in the original processor-flow graph. From the reduced number of possibly low cost assignments the use of indices enables INDEX-I to find the optimum solution in 77% of the cases and a near optimum solution in the remainder.

Turning to INDEX-G, we note that it is not assisted by any reduction in the number of assignments. In view of the large number of possible assignments with which INDEX-G is faced, its performance is surprisingly good. The number of possible assignments that assign \( n_T \) nodes to \( T \) in an \( n \)-node graph,

\[
\begin{align*}
N(n_T) &= \binom{n}{n_T} \\
&= \frac{n!}{(n-n_T)! \cdot n_T!}
\end{align*}
\]

\( N(n_T) \) grows rapidly as \( n_T \) increases. \( N(0) = 1, N(1) = n, N(2) = n^2/2, \ldots \). For the smallest graph size considered, \( n = 32 \), even the third term is very large. INDEX-G
examines one cut for each value of \( n_T \). Hence the fraction of all possible cuts of a given size that it examines,

\[
f(n_T) = \frac{1}{N(n_T)}
\]

Clearly for \( n \geq 32 \) and \( n_T \geq 2 \), \( f(n_T) \) is extremely small. In addition, the expected cost of an arbitrary cut increases with \( n_T \) ((3.6), (3.6a), Fig. 3.3). Hence we might expect that the solutions found by INDEX-G would have \( n_T = 0 \) or 1 in most cases. Table 4.4 shows the mean value of \( n_T \) for different sizes of input for INDEX-G. This is plotted in Fig. 4.7. We see that in all cases except \( n = 32 \), the mean value of \( n_T \) is greater than 3, increasing to nearly 8 in the case \( n = 82 \). This leads us to hypothesize that the use of indices enables INDEX-G to identify localities of modules and thus perform well.

For comparison Table 4.4 shows the mean value of \( n_T \) for the optimum solution. It is consistently higher than that for INDEX-G. We further hypothesize that the indices enable INDEX-G to identify only a few localities. This would also help explain the fact that the performance of INDEX-G slowly worsens as \( n \) increases (Figs. 4.6a, 4.6b).

We now discuss the significance of the better performance of INDEX-I over INDEX-G. In a practical situation costs may not be known with any great degree of precision. Hence a 9% improvement may not be significant, especially in view of the faster execution of INDEX-G. However, these scheduling techniques are viable only for production
Table 4.4 $\bar{n}_T$ for INDEX-G and the minimum feasible cut.

<table>
<thead>
<tr>
<th>Size of Input, $n$</th>
<th>Sample Size</th>
<th>Mean number of modules Assigned to $T$, $\bar{n}_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>INDEX-G</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>0.7</td>
</tr>
<tr>
<td>42</td>
<td>10</td>
<td>3.2</td>
</tr>
<tr>
<td>52</td>
<td>10</td>
<td>4.3</td>
</tr>
<tr>
<td>62</td>
<td>10</td>
<td>5.6</td>
</tr>
<tr>
<td>72</td>
<td>10</td>
<td>4.9</td>
</tr>
<tr>
<td>82</td>
<td>6</td>
<td>7.8</td>
</tr>
</tbody>
</table>
Fig. 4.7  Mean number of modules assigned to T versus n.
programs which are executed often. Over a period of time the recurring saving due to the lower cost of the assignment found by INDEX-I may justify the extra computation required initially. Another case in which INDEX-I may be useful is when we desire to find assignments for a range of memory capacities on T. Since the ICG need be computed only once, the execution time for INDEX-I will increase marginally while that for INDEX-G will increase substantially. These are issues that can be meaningfully addressed only when there exists a sufficient base of data on programs written for distributed processor systems and in the context of particular systems.

4.4 **SUMMARY.**

In this chapter we have described two heuristics for limited memory scheduling – INDEX-I and INDEX-G. The former uses the ICG while the latter uses only the processor-flow graph. We have established polynomial bounds on their worst-case time complexities. For constant degree graphs, based on simulation and the results of chapter 3 we have investigated and compared their performances. We have concluded that subject to the limitations of the model the heuristics have merit in certain cases. INDEX-I is slower than INDEX-G but performs better. Unless costs are known with some degree of precision this improvement may not be significant.
CHAPTER 5. CRITICAL LOAD FACTORS AND THE ICG.

In this chapter we show a connection between two problems described in chapter 2 - scheduling with limited memory and scheduling with variable load factor. In particular we show that if nodes can be condensed in the ICG then they have the same critical load factor. This relationship is of interest because it enables us to apply certain results from one problem to the other.

Assume that the load factor $f$ on processor $T$ is variable while that on $S$ is constant. With $f = 1$ compute the ICG for reassigning nodes from $T$ to $S$. Theorem 5.1 establishes that all the nodes in a condensed node of the ICG have the same critical load factor. We attempt first to give an intuitive proof. Refering to Fig. 5.1 let cut I be the mincut at $f = 1$ and assume that $C$ and $D$ are condensed in the ICG. That is, if $C$ is reassigned to $S$ (by setting $c(S,C)$ to infinity) then cut II is the mincut and similarly if $D$ is reassigned to $S$ then cut II is still the mincut.

Let $f$ decrease continuously from 1. The costs $c(S,C)$ and $c(S,D)$ increase. Thus the cost of cut I increases. At some value $f_1$ of $f$ let some other cut equal cut I in cost and let this cut have a lower cost for $f < f_1$. Thus one or both of $C$ and $D$ migrate to $S$ at $f = f_1$. There are three possibilities for the new mincut - cuts II, III or
Fig. 5.1  Simple Processor-flow Graph.
IV. If II is the new mincut the theorem holds. Assume that III is the new mincut. We know that at $f = 1$ the value of cut III is no less than that of cut II, for otherwise III would be the mincut reassigning D to S. As $f$ decreases to $f_1$ the cost of cut II does not change while that of cut III increases. Thus III cannot be less than II at $f = f_1$. By a similar argument IV also cannot be less than II in cost at $f = f_1$. Hence C and D migrate together and have the same critical load factor.

On these lines we now present a formal proof for the general case. We use the proof techniques of Gomory and Hu [1961].

**Theorem 5.1.**

" In the 2-processor distributed system described above compute the ICG for reassigning nodes from T to S with $f = 1$. For each condensed node in the ICG, every node in the condensed node has the same critical load factor."

**Proof.**

Let $C(X,Y)$ denote the total weight on all branches between subsets $X$ and $Y$ if this is a constant with respect to load factor. If the total branch weight is a function of $f$, let $F_f(X,Y)$ denote the value at load factor $f$.

Refering to Fig. 5.2, let $\{C,D\}$ be a condensed node in the ICG, where C and D are non-empty subsets of nodes.
Fig. 5.2 General Processor-flow Graph.

Processor-flow Graph.

ICG at f=1.

Fig. 5.3 Counter-example to the converse of Thm. 5.1.
Let cut I be the mincut reassigning \{C,D\} to S at \( f = 1 \).

At some load factor \( f_1 \), assume that the mincut separating S and T is cut II. Thus the critical load factor of \{C\} is \( f_1 \) and that of \{D\} is less than \( f_1 \). Note that either or both of the subsets \{B\} and \{E\} may be empty.

The weight of cut II is no greater than that of cut II re-routed to assign \{B,D\} to S at \( f = f_1 \).

\[
i.e. \quad F_{f_1}(A,B) + F_{f_1}(A,D) + \ldots \leq \begin{cases} F_{f_1}(A,F) + C(B,F) + C(C,D) + C(C,F) + 1 \\ \ldots \end{cases} \quad (1)
\]

Simplifying,

\[
i.e. \quad F_{f_1}(A,B) + F_{f_1}(A,D) + \ldots \leq C(B,F) + C(D,F) \quad (2)
\]

\{C,D\} is a condensed node of the ICG. Hence the weight of cut I is no greater than that of cut I re-routed to assign \{B,D\} to T, with \( f = 1 \).

\[
i.e. \quad F_1(A,E) + F_1(A,F) + \ldots \leq \begin{cases} F_1(A,B) + F_1(A,D) + \ldots \\ \ldots \end{cases} \quad (3)
\]
or, \( C(B,E) + C(B,F) + C(D,E) + C(D,F) \) \( \leq \) \( F_1(A,B) + F_1(A,D) + C(B,C) + C(C,D) \)

Since costs are non-negative this inequality holds if we subtract \( (C(B,E) + C(D,E)) \) from the LHS and add it to the RHS. Also \( F_1(X,Y) \leq F_\leq(X,Y) \), for all \( X, Y \) and \( f_\leq \leq 1 \). Thus (4) becomes:

\[
\begin{align*}
C(B,F) + C(D,F) & \leq C(B,E) + C(D,E) + (5) \\
F_1(A,B) + F_1(A,D) + C(B,C) + C(C,D) & \leq (4)
\end{align*}
\]

(2) and (5) can hold simultaneously only with equality. Hence (1) is an equality and the weight of cut II rerouted to assign \( \{B,D\} \) to \( S \) is equal to the weight of cut II. Thus all the nodes in \( \{C,D\} \) have the same critical load factor.

Q.E.D.

Figure 5.3 shows a case for which the converse of the above theorem is not true. Nodes A and B have the same critical load factor of \( 12/13 \) but are not condensed in the ICG.

Corollary 5.1
"The number of nodes in the ICG is an upper bound on the number of module clusters with the same critical load factor."

One implication of Theorem 5.1 is that the polynomial-
time algorithm of Michel and van Dam [1977] for computing critical load factors can be used as a heuristic for the limited memory scheduling problem. The algorithm finds the mincuts at which the assignment changes as \( f \) varies from 1 to 0. As a consequence of Theorem 5.1 these assignments correspond to cuts in the ICG. By the inclusion property of the critical load factors these cuts are ordered on the total size of modules assigned to \( T \). We can take the minimum valued feasible cut from among these as the solution. Alternately, we can find the \( i \) such that cut \( i \) is feasible and cut \( (i+1) \) is not feasible. We can then find the minimum of all feasible cuts between these two as the solution. We have not investigated either heuristic.

As a result of corollary 5.1 results on the amount of condensation identified by the ICG can be applied to variable load factor clustering. Thus, for the instances we used in section 3.2 the mean number of clusters is no greater than the mean size of the ICG. This indicates that for constant connectivity and constant degree graphs with the parameters used modules break up into relatively few clusters under variable load factor.

However, due to the fact that the theorem holds in one direction only caution must be exercised. For example, the critical load factors can be computed in polynomial-time but the ICG requires exponential-time in the worst
5.1 **SUMMARY.**

In this chapter we have proved that nodes in a condensed subset of the ICG have the same critical load factor under variable load factor. We have indicated possible applications of this result.
CHAPTER 6.  CONCLUSION.

We first summarize the thesis and then present suggestions for further work in this area.

6.1 SUMMARY.

This thesis has studied heuristic algorithms for 2-processor scheduling with limited memory. We have first shown that obtaining an absolute approximation solution is as hard as obtaining an exact solution. This has motivated us to look for efficient heuristics which may perform poorly in some instances. Next we have presented statistics indicating the reduction possible in the ICG for two classes of problem instances. This has motivated the development of a heuristic based on the ICG. Then we have established asymptotic bounds on the expected value of the minimum feasible assignment for limited memory scheduling, subject to certain restrictions.

These preliminaries have lead to the main result of the thesis - the development, analysis and testing of two polynomial-time heuristics for limited memory scheduling. We have shown that subject to validation of our models these heuristics are viable tools for limited memory scheduling.

An interesting ancillary result is a relationship
between limited memory scheduling and variable load factor scheduling. We have discussed some of the implications of this restricted equivalence.

6.2 **SUGGESTIONS FOR FURTHER WORK.**

An issue that needs to be settled is the existence of a polynomial-time \(\epsilon\)-approximation algorithm for limited memory scheduling. If such an algorithm does not exist then it may be profitable studying algorithms that are "good" in some probabilistic sense (Karp [1976]).

Before this thesis and other work in this area can be related to practical systems, detailed analyses of classes of programs are necessary. Among issues to be studied are the distributions of costs and the existence of locality in intermodule reference.

Further constraints can be introduced, eg., limited memory on one processor and variable load factor on the other. A consideration on concurrency of execution may require development of a different model.

Finally the general case of \(n\) processors can be considered. Stone [1977, 1977a] has extended the flow-graph model to \(n\)-processor systems. For the unconstrained 3-processor case he has an algorithm which finds an optimum solution in many cases. For \(n > 4\) the problem has been shown to be NP-complete. Constrained problems may be even harder. It
may well be that the flow-graph model is not suitable for the general case and more powerful tools may have to be developed to attack the problem. Here too probabilistically "good" algorithms may be useful.
REFERENCES.


Michel, Janet and Andries van Dam, [1977], "Evaluation of Performance Improvement in Distributed Processing", draft, Program in Computer Science, Brown University, Providence, RI, 18 pp, Aug. 1977.


