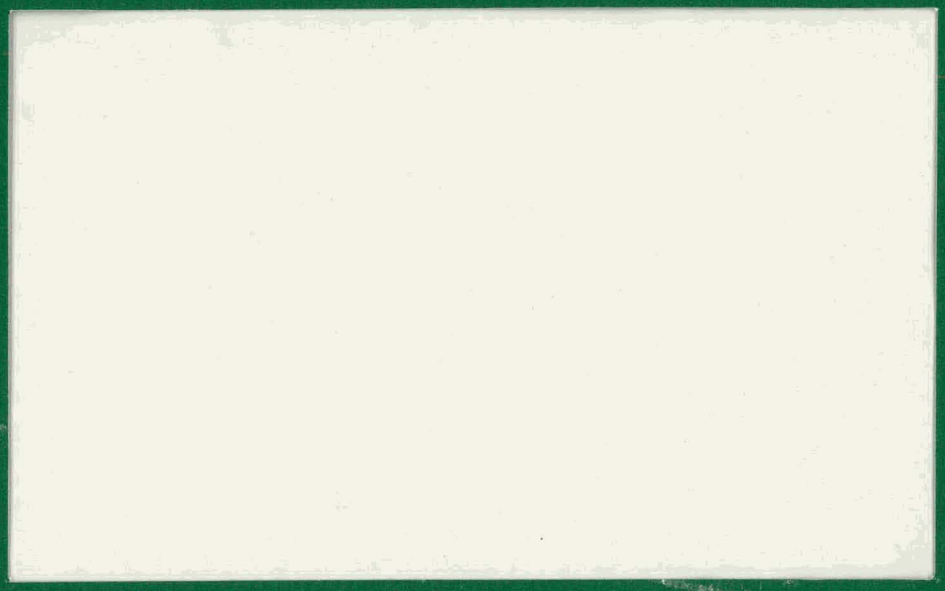
*Center for Supercomputing Research & Development*





*Center for Supercomputing Research & Development*
*National Center for Supercomputing Applications*
University of Illinois at Urbana-Champaign

## DISCLAIMER

CSRD Rpt. No. 656

UILU-ENG-87-8002

DOE/ER/25001--45

DE88 003598

# NONPREEMPTIVE RUN–TIME SCHEDULING ISSUES ON A MULTITASKED, MULTIPROGRAMMED MULTIPROCESSOR WITH DEPENDENCIES, BIDIMENSIONAL TASKS, FOLDING AND DYNAMIC GRAPHS

Allan Ray Miller

May 1987



Center for Supercomputing Research and Development
University of Illinois
305 Talbot – 104 South Wright Street
Urbana, IL 61801–2932
Phone: (217) 333–6223

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84–10110 and US NSF DCR84–06916, the U. S. Department of Energy under Grant No. US DOE–DE–FG02–85ER25001, the IBM Donation, and the Control Data Corporation, and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science, May 1987.

# NONPREEMPTIVE RUN-TIME SCHEDULING ISSUES ON A MULTITASKED, MULTIPROGRAMMED MULTIPROCESSOR WITH DEPENDENCIES, BIDIMENSIONAL TASKS, FOLDING, AND DYNAMIC GRAPHS

BY

ALLAN RAY MILLER

B.S., University of Central Florida, 1979
M.S., University of Illinois, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

# NONPREEMPTIVE RUN-TIME SCHEDULING ISSUES ON A MULTITASKED, MULTIPROGRAMMED MULTIPROCESSOR WITH DEPENDENCIES, BIDIMENSIONAL TASKS, FOLDING, AND DYNAMIC GRAPHS

Allan Ray Miller, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1987
Duncan H. Lawrie, Advisor

Increases in high speed hardware have mandated studies in software techniques to exploit the parallel capabilities. This thesis examines the effects a run-time scheduler has on a multiprocessor. The model consists of directed, acyclic graphs, generated from serial FORTRAN benchmark programs by the parallel compiler Parafrase. A multitasked, multiprogrammed environment is created. Dependencies are generated by the compiler. Tasks are bidimensional, i.e., they may specify both time and processor requests. Processor requests may be folded into execution time by the scheduler. The graphs may arrive at arbitrary time intervals. The general case is NP-hard, thus, a variety of heuristics are examined by a simulator. Multiprogramming demonstrates a greater need for a run-time scheduler than does monoprogramming for a variety of reasons, e.g., greater stress on the processors, a larger number of independent control paths, more variety in the task parameters, etc. The dynamic critical path series of algorithms perform well. Dynamic critical volume did not add much. Unfortunately, dynamic critical path maximizes turnaround time as well as throughput. Two schedulers are presented which balance throughput and turnaround time. The first requires classification of jobs by type; the second requires selection of a ratio value which is dependent upon system parameters.

# ACKNOWLEDGEMENTS

First I'd like to thank my advisor, Professor Duncan Lawrie, who gave me much assistance, and gave me the privilege and opportunity of working at CSRD. Also, I'd like to acknowledge those who have supported me while at this university, i.e., the Center for Supercomputing Research and Development, the Department of Computer Science, the National Science Foundation, the United States Department of Energy, IBM, and Control Data Corporation. Finally, since the time of elementary school flashcards, my parents taught me the value of education, and provided me with essential love and support. I have come this far because of them.

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1. High Speed Hardware and Software

In 1944 the MARK 1 computer required 300 milliseconds to complete an addition operation. By 1970 the ILLIAC IV performed 50 million floating point operations per second (megaflops). Current supercomputers are capable of performing at peak speeds approaching one gigaflops.

Originally, computers were becoming faster mainly due to improvements in device technology. Such technology improvement, however, is ultimately limited by the speed of light. Naturally then, research has turned to parallel hardware and software in order to continue the growth in computer speed.

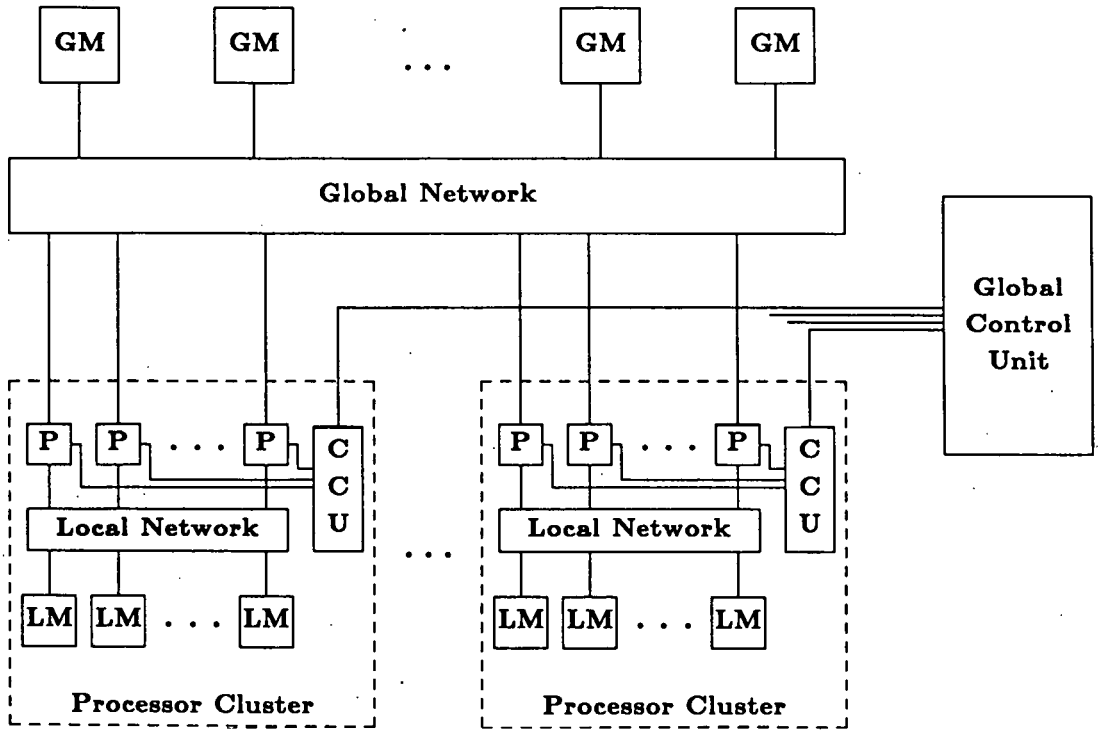
Many of the early supercomputers, such as the Cray 1, employed vector hardware in order to achieve their high speeds. Since scientific calculations often contain substantial vector arithmetic, these systems include special vector instructions. By subdividing operations into sequential and separate suboperations, vector processors can pipcline successive vector elements through the hardware and obtain a high degree of concurrency.

While vector processors have been very successful, they do have certain hardware and software limitations. For example, the speedup pipelined hardware can offer is limited by the number of stages in the pipe. That is, any particular

operation can only be subdivided so far. Another inherent problem with pipelined architectures is the overhead involved with initially configuring the pipes, and then filling them with data before any results emerge. Because of this problem, speedup grows very slowly, typically requiring vectors of length 100 or greater in order to achieve peak performance. Finally, these systems require appropriate software in order to successfully exploit the hardware. Either the programming language must allow the programmer to express his algorithm in a vector form, or else some sort of restructuring compiler will be required to generate the vector instructions from a traditional scalar language.

Much of the current supercomputer research has focused upon multiprocessing systems. For example, the Cedar supercomputer [KDLS86] [GKLS83] as shown in Figure 1.1, is under development at the Center for Supercomputing Research and Development. Also, commercially available systems such as the Cray X–MP are already on the market.

Multiprocessors achieve their speed by doing several (possibly different) operations in parallel. Whereas vector processor speedup is constrained by the number of subdivisions in the pipes, parallel processors can (to some extent) double their peak performance simply by doubling the number of processors available for computation. This does not mean that these two concepts are mutually exclusive. Indeed, most current parallel processing architectures employ various types of pipelining throughout the system. But the big advantage of multiprocessors is that their architecture can be scaled up across time for a continued growth in speed.




P: Processor
GM: Global Memory Module
LM: Local Memory Module
CCU: Cluster Control Unit

Figure 1.1. Cedar Architecture.

Multiprocessors also require extensive software support in order to maximize their performance by completely exploiting the available hardware. Synchronization and data coherency problems must be dealt with. And like vector processors, they require parallel programming languages or restructuring compilers capable of recog-

nizing parallel constructs.

Generally, three levels of parallelism can be recognized in the software. The highest level involves separate programs, subroutines, loops, or control flow paths within the same program. The next level down consists of parallelism between individual loop iterations, which can be assigned to separate processors. The potential for speedup at this level is enormous. It is often proportional to the loop limit, or the product of nested loop limits. And finally, the lowest level of parallelism which can be exploited by multiple processors is the separate arithmetic expressions in individual assignment statements. Such low level parallelism offers the least potential for high speedup, however.

Once the algorithm has been designed, the parallel constructs recognized and the code written, the parallel code constructs must be scheduled for execution on the processors. Such scheduling can be performed at compile time, run time, or a combination of both [Poly86]. Certainly some of the scheduling must be done at run time, as the presence of other jobs in the system competing for resources may affect the scheduler's decision as to which subsection of code is the best candidate for execution.

There are several factors a run–time scheduler must consider when deciding which section of code should be initiated. For example, the presence of data dependencies may force a partial ordering upon the code. The number of processors requested by each unit of code in light of how many processors are currently available is obviously an important consideration. Tradeoffs between speed (execution

time) and size (number of processors allocated by each task) must be made. New programs or sections of programs may continually be arriving in the system at a variety of different times. The scheduler may want to maximize hardware utilization, throughput, or minimize turnaround time for each program. When these and many other factors must be sifted through for each of the thousands or tens of thousands of code segments wishing service from the processors, it clearly is a complicated decision which awaits any run–time scheduler. It is this line of research that this thesis will attempt to investigate.

### 1.2. Related Work

A lot of similar work has already been performed in the multiserver scheduling and operations research areas. Virtually all of it, however, has dealt with abstract analytical models with limited practical significance. This thesis attempts to see how well a variety of scheduling techniques perform on real benchmark programs generated by a compiler. Not only are real programs the things which ultimately must be scheduled on functioning multiprocessors, but unlike theoretical models, their characteristics cannot be determined *a priori*.

This work investigates several extensions over previous results. For example many models restrict themselves to two processor systems [Ston78] [RaSH79] [Bokh79] [CHLE80] whereas this model may have any number of processors desired.

Data dependencies in the form of a directed acyclic graph exist between tasks which the scheduler must enforce. Many models ignore any possible data dependen-

cies such as [Stan85] [ChAb82]. Some previous work has examined this problem, e.g., [ChTs81] [Ston77] [StBo78] examines a very simple sequential chain model, while the models of [Kras72] [RaCG72] [HoIr83] [Schw61] do include complete graphs. Those models, however, make simplifying assumptions in other areas which this thesis does not make. Furthermore, this model assumes that the graphs may be dynamically updated with new node and dependence arc arrivals as additional program and program subsections are added to the system.

Another feature of this model is that each node may request multiple processors as well as multiple time units needed for completion of the task. Nodes are thus two dimensional. Several models feature $p$ processor systems [CeKl83] [AbDa86] [ChKo79] [KrWe85] although each task requests only one of the processors. In [BlDW86] tasks request multiple processors, but have unit times. Some models have allowed variable time requirements [Kras72] [RaCG72] but then have a unit processor request.

Not only have bidimensional nodes been allowed, but processors can fold into time by running parallel constructs serially. In [XuYe83] [XuYe84] [Mill84] this concept was studied, although again simplifying assumptions in other areas of the models were made. (Folding is also discussed in [Sahn83] as it relates to pipelined machines).

Closely related work to that done in this thesis was performed by Hu [Hu61] [Hu82]. Hu's model allows data dependencies in the form of a tree. The first phase of his algorithm determines the distance from the root of the tree to each node.

Next, the scheduler simply assigns nodes to the processors starting at the highest levels of the tree first and working its way down towards the root.

Hu calls this algorithm the *critical path algorithm* and it is optimal. Its limitations arise from the fact that nodes are nondimensional (unit time and processor requirements), must be completely known in advance, and must be in the form of a tree. However, two modifications of his algorithm which incorporate several extensions to that model will be presented later in this thesis. Some of the same concepts were developed independently in [Poly86] for compile time scheduling.

Many of the other scheduling models are also optimal [Coff76] [CoGr72] given their assumptions of at most two processors, unit time or processor requirements, etc. Such an optimal algorithm is not likely for the model studied in this thesis. For example, the simple extension of variable processor requirements in Hu's work moves it from the domain of a very fast optimal algorithm to the bin packing problem which is NP–hard [GaJo79]. But this thesis goes beyond even that, creating a two dimensional bin packing problem with folding, dependencies, and dynamically modified graphs.

It is clear that unless many simplifying assumptions are made, an optimal run–time scheduler cannot be found. Nevertheless, the problem must be examined, as multiprocessors do exist which need the services of some scheduling heuristic in real time. Failure to do so will result in an inefficient exploitation of the parallel hardware by the software which runs on it. This area will be the focus of the following work.

## 1.3. Thesis Overview

This thesis attempts to solve some small portion of the run–time scheduling problem on a multiprogrammed, multitasked, multiprocessing computer with data dependencies, bidimensional nodes, folding, and dynamic graphs. The scheduler will examine data taken from real benchmark programs.

This chapter has already provided an introduction and a discussion of related work. Chapter two defines the basic model and provides the framework of the discussion. It, along with the Appendix, describes the software used to investigate the problem. Chapter three discusses the data and scheduling algorithms used in the simulations. Chapter four presents and analyzes the results of the simulations, and trys to demonstrate why some schedulers perform better than others. Certain tradeoffs must be made, and they are discussed in this chapter. Finally, Chapter five summarizes the results and draws the conclusions. It also discusses limitations of this study, and suggests areas of future research.

# CHAPTER 2

# BASIC CONCEPTS AND MODELS

## 2.1. Parafrase

Before parallel code constructs can be scheduled for execution on the processors, they first must be generated, either by a parallel programming language, or by a restructuring compiler. The way this problem has been solved for purposes of this thesis is with the Parafrase restructuring compiler, which has been under development at the University of Illinois for the last fifteen years [KKLW80] [KKPL81] [Wolf82]. The overall structure of Parafrase is shown in Figure 2.1.

Parafrase was chosen because of the large investment in existing sequential FORTRAN software. Also, FORTRAN (in some form) is the most widely used high level language on supercomputers [PeZa86]. Furthermore, it is not clear that programmers can deal with the complexity of parallel hardware and software on average as well as a compiler.

Parafrase starts by reading sequential FORTRAN programs. It then performs a series of standard transformations, or passes, on the source code. After each pass, various data structures and a modified FORTRAN source program with parallel constructs are produced. By modifying which passes are called, the programmer can control such attributes as specific optimizations for the particular target architecture, and many other features.

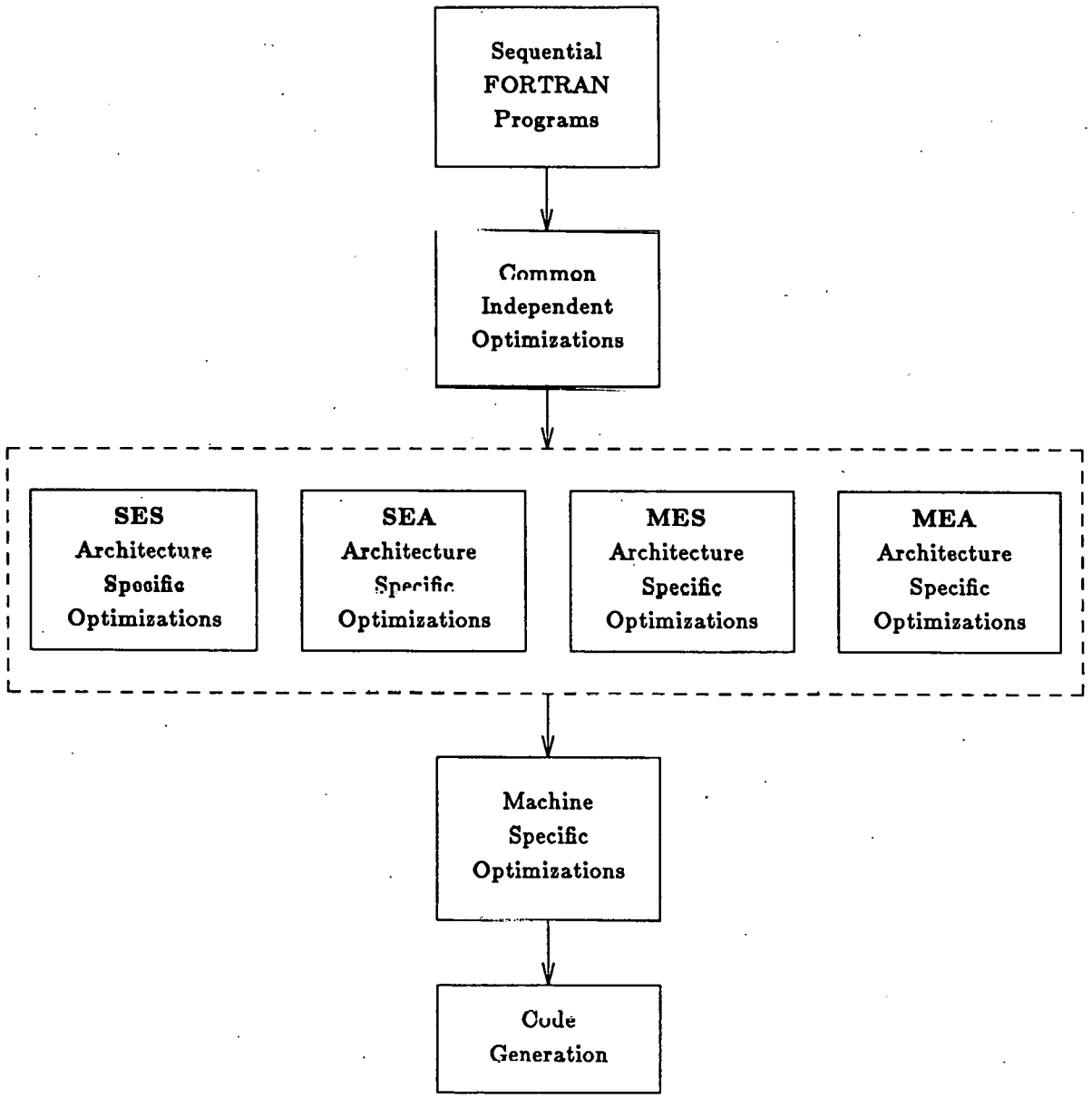



Figure 2.1. Overall Structure of Parafrase.

Critical to the Parafrase compiler, as well as to any run–time scheduler which must examine the output of Parafrase, is the notion of *data dependencies* [Kuck78]. Parafrase is capable of recognizing three different forms of data dependencies. Assume some program contains two different statements, $S_i$ and $S_j$ such that $S_j$ follows $S_i$ during serial program execution. If a variable $x$ is assigned in statement $S_i$, and subsequently used on the right hand side of statement $S_j$, then $S_j$ is said to be *data flow dependent* on $S_i$. If the variable $x$ is used on the right hand side of $S_i$ but is assigned in $S_j$, then $S_i$ is *data antidependent* on $S_j$. If $x$ is assigned in both $S_i$ and subsequently in $S_j$, then $S_j$ is *data output dependent* on $S_i$. Finally, Parafrase will introduce *control dependencies* induced by the presence of conditional statements.

Parafrase analyzes the FORTRAN source program, and generates a data dependence graph between the statements. It is then capable of performing sophisticated transformations on the results which break many of the control and data dependencies without violating the semantics of the program. Any dependence arcs which remain after all of the transformation and optimization passes have completed, force a partial ordering upon the program execution, which must of course be honored by the run–time scheduler.

At the current time, Parafrase has over 50 different passes which can be called, performing such transformations as forward statement substitution, loop interchanging, recurrence relation recognition, etc. The specific passes which are called and the order in which they occur are a function of the particular architecture which is to be targeted.

Parafrase recognizes four classes of machines: Single Execution Scalar (SES), Single Execution Array (SEA), Multiple Execution Scalar (MES), and Multiple Execution Array (MEA). SES machines are simple uniprocessors. SEA machines include array and vector processors. MES machines are multiprocessors where each processor is composed of a SES machine. And finally, MEA machines are multiprocessors where each processor is composed of a SEA machine. The model chosen for study in this thesis would be applicable to either a MES or a MEA machine.

```
C   COUNTS NUMBERS OF PARTITIONS OF AN INTEGER
    SUBROUTINE COUNT (C, K, P, N)
    INTEGER C, P
    DIMENSION C(K), P(N)
    DO 10 I = 1, N
      P(I) = 0
10  CONTINUE
    DO 30 I = 1, K
      J = C(I)
      JP1 = J + 1
      P(J) = P(J) + 1
      DO 20 M = JP1, N
        MMJ = M - J
        P(M) = P(M) + P(MMJ)
20    CONTINUE
30  CONTINUE
    RETURN
    END
```

Figure 2.2. Subroutine COUNT.

Consider the short subroutine COUNT shown in Figure 2.2. Although it is much too simple to illustrate most of the features of Parafrase, it will serve, to some extent, to demonstrate how data can be generated from real FORTRAN programs for a run–time scheduling simulation.

Midway through the series of passes selected for use for this thesis, intermediate results were produced by Parafrase as shown in Figure 2.3. Two short parallel loops have been generated; they are flagged with an asterisk. Each iteration of the parallel loops may be scheduled simultaneously for execution on separate processors. (They

```
1                       SUBROUTINE COUNT (C, K, P, N)
2                       INTEGER C(K), I, J, J'(*), JP1, K, M, MMJ, N, P(N)
3          +--          DO 1 i = 1, N
4          *              P(i) = 0
5          +--      1   CONTINUE
6          +--          DO 2 i = 1, K
7          *              J'(i) = C(i)
8          +--      2   CONTINUE
9       +------         DO 4 i = 1, K
10      |                 P(J'(i)) = 1 + P(J'(i))
11      |  +--           DO 3 j = 1, N - J'(i)
12      |  |              P(j + J'(i)) = P(j + J'(i)) + P(j)
13      |  +--      3    CONTINUE
14      +------    4    CONTINUE
15                      RETURN
16                      END
```

Figure 2.3. Intermediate Results of COUNT.

can of course be executed sequentially, should insufficient processors be available at run time). Parafrase also generated two nested serial loops from COUNT. These two loops are flagged with vertical bars. The problem occurs because the assignment of the array P in Figure 2.2 is essentially being subscripted by the value of the array C. Figure 2.3 clearly shows the subscripted subscripts hence, these two loops cannot be sped up. Unlike the parallel loops, this portion of the program must be scheduled sequentially for execution.

The next series of passes performed high level spreading and compound function generation on COUNT. This will be discussed in the next section.

## 2.2. Compound Functions and Program Graphs

The unit of work which is to be scheduled on the multiprocessor is referred to as a *compound function* or *task* [GLPV83] [KLVY82] [Husm86]. Although there are many ways to construct and define a task, for purposes of this thesis a task is defined to be some portion of a larger program which can be scheduled for execution for a fixed amount of time on a fixed number of processors. (The execution time and number of processors requested are fixed, or bound, at allocation time. This will be discussed in more detail below.) Once a task has begun execution, it runs to completion without interruption. That is, tasks are said to be *nonpreemptive* for purposes of this thesis.

The model chosen for study receives as input directed acyclic graphs (DAGs) such as the one shown in Figure 2.4. Each node in the graph represents a task which

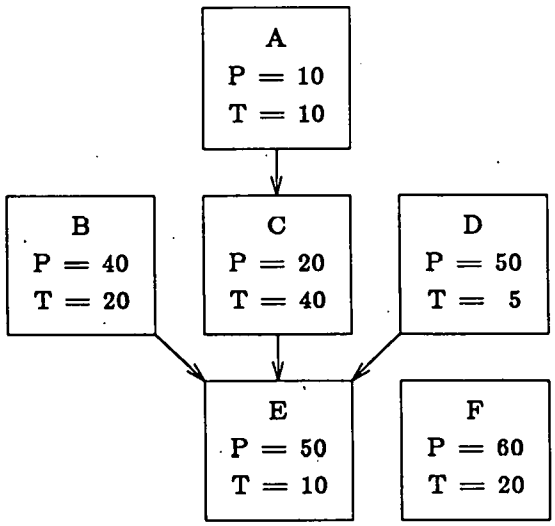



Figure 2.4. Program Execution Graph.

is to be scheduled for execution on the processors. The nodes contain two numbers: the maximum number of processors needed by that task, and the execution time that task requires in order to complete its work. For example node B, or $N_B$, in Figure 2.4 requests 40 processors which it intends to reserve for 20 units of time.

It is important to note that nodes are *bidimensional* in that their space–time products on the processors can have nonunit values in both directions. Furthermore, a one–way mapping from processors into time is possible (but not the other way around). No data dependencies exist between those operations, or else they would have been assigned to different time slices at an earlier stage (by the compiler).

Clearly then, if no dependencies exist between the parallel operations, they can be run serially as well as in parallel without violating the algorithm's semantics. Such a concept is known as *folding.*

It is very useful to give the run–time scheduler the ability to dynamically fold the tasks at processor allocation time. For example, if a task requests $p$ processors, but only $p/2$ processors are currently available (due to other tasks in the system competing for resources) the scheduler can fold task execution across $p/2$ processors simply by doubling its running time. However, such a scheduler must be very careful. Too little folding results in idle processors while large tasks are forced to wait for a sufficient number of processors to become available. This can lead to a serious degradation in performance as well as utilization. On the other hand, too much folding can also lead to loss of speedup as a result of parallel constructs being forced to run serially. In [XuYe83] [XuYe84] it was shown that on the average, if at least 25 percent of the processors requested by a task are currently available, then that task should be folded and initiated at once.

Control or data dependencies in a graph passed on from the compiler force a partial ordering upon a scheduler which it must honor. For example, if $N_A$ must complete its execution before $N_C$ can begin, then it is said that $N_A > N_C$ and a directed arc is drawn from $N_A$ to $N_C$ in Figure 2.4.

In this scheduling model, the program graphs must be acyclic. Innermost serial loops can easily be transformed into a single task. However, outer level nested serial loops must be *unrolled* and the associated nodes and arcs replicated up to the loop

bounds.

Conditional paths through the graph are treated in the same fashion as what Parafrase does. For example, Parafrase attempts to transform backwards GOTOs into DO loops, which can then be treated in the normal fashion. Other conditional paths with control dependencies which cannot be eliminated are expanded and *weighted* according to the probability of that path being chosen. (The default probabilities are to set all paths as being equally likely). The weighted nodes and their associated arcs are then entered into the program graph.

The tasks, represented by nodes in the DAG, are defined to be *dynamic*. That is, they may arrive at various times throughout the execution of the programs. This means that the scheduler only sees a snapshot of the entire graph at any one time. Since the introduction of new nodes with their dependence arcs into the system may affect which tasks are the best candidates to dispatch for execution, the run–time scheduler must resign itself to making decisions based only upon a particular instance of the entire DAG.

Now that the particular form of tasks and DAGs used in this thesis has been defined, it remains to be shown how the actual tasks were generated by Parafrase for use in the scheduling simulation model. Specifically, the tasks were derived from a method called *high level spreading* described by Veidenbaum in [Veid85].

Veidenbaum first decomposes the program into a number of high level objects (HLOs). He defines eight types of HLOs: **1)** a nonnested (innermost) DO loop of any type of parallelism, together with all of the statements in it; **2)** any Block of

Assignment Statements (BAS) not in 1). (A BAS is the largest possible block of consecutive assignment statements in the serial program with one entry and one exit point); **3)** a nested DO statement not in 1) but without statements inside; **4)** any IF statement not in 1); **5)** the CONTINUE statement for nested DOs and for IFs (the terminal line); **6)** user procedures or function calls. (Veidenbaum assumes subroutine expansion was used to eliminate these in innermost loops); **7)** I/O statements; and finally **8)** program/subroutine/function BEGIN and END statements.

Returning to the previous example of COUNT in Figures 2.2 and 2.3, the high level spreading passes in Parafrase produced the results seen in Table 2.1. The compound function numbers are denoted, along with any nested or successor compound functions which may be associated with that compound function. The successor pointers will be utilized to generate the dependence arcs for the DAG. The "first" and "last" columns denote the first and last statement numbers in Figure 2.3 which were used in the generation of that particular compound function. The "type"

Table 2.1. High Level Spreading of COUNT.

| CF # | Nested | Successors | First | Last | CF Type | Processors | Time |
|---|---|---|---|---|---|---|---|
| 1 | 2 | – | 1 | 1 | C_PROG | 80 | 3281 |
| 2 | 0 | 4 | 3 | 5 | E_DO | 40 | 1 |
| 3 | 0 | 4 | 6 | 8 | E_DO | 40 | 1 |
| 4 | 5 | 7 | 9 | 9 | C_DO | 1 | 3280 |
| 5 | 0 | 6 | 10 | 10 | BAS | 1 | 2 |
| 6 | 0 | – | 11 | 13 | E_DO | 1 | 80 |
| 7 | 0 | 8 | 14 | 14 | C_DOEND | 0 | 0 |
| 8 | 0 | 9 | 15 | 15 | C_RETRN | 0 | 0 |
| 9 | 0 | – | 16 | 16 | C_END | 0 | 0 |

column indicates what form of compound function is listed. C_PROG and C_END denote the beginning and end of a program. C_RETRN denotes a subroutine return statement. C_DO and C_DOEND denote an outer nested loop. E_DO denotes an innermost loop and all statements within that loop. And finally, BAS denotes a block of assignment statements. The processor and time columns represent the number of processors requested and the times those compound functions require (or some calculated estimate if compound functions are nested within).

Comparing Figure 2.3 with Table 2.1, it can be seen that the two parallel loops were transformed into E_DO compound function numbers 2 and 3. Since they are a simple assignment statement, they only take one unit of time. Parafrase scheduled these loops on 40 processors. If loops bounds are known at compile time, then Parafrase will use those values. Otherwise, Parafrase uses a default timing value of 40 for all loop indexes in order to make some estimates of the work involved.

The innermost nested serial loop was turned into E_DO number 6. Each iteration takes two units of time (an addition plus the assignment). Since this only requires one processor, but must be replicated 40 times, the E_DO is flagged for 80 time units on a single processor. The outermost nested serial loop contains a similar arithmetic statement, along with the inner E_DO. For identical reasons, it also only requests a single processor. Its time requirements reflect the work nested inside the loop, replicated serially 40 times.

Graphically, Table 2.1 can be represented by Figure 2.5. The vertical bracket represents sequential replication. Clearly, $N_A > N_C$ and $N_B > N_C$. Furthermore,

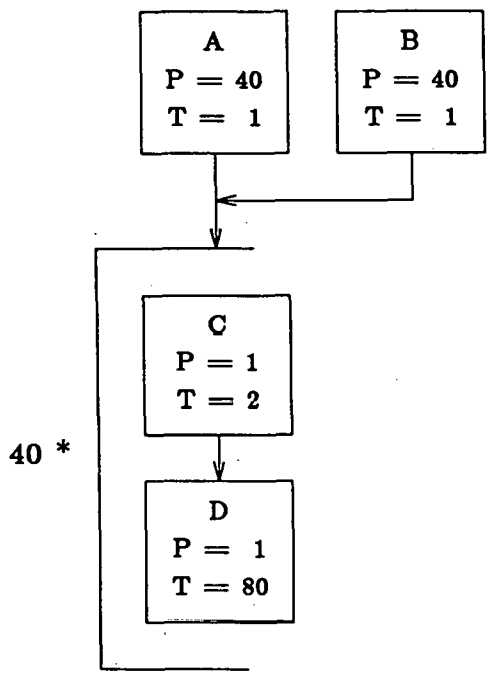



Figure 2.5. Graphical Representation of COUNT.

between each iteration $i$ of the outer serial loop, $N_{C_i} > N_{D_i}$ and $N_{D_i} > N_{C_{i+1}}$.

It should be clear how the DAG used for scheduling could be generated from Figure 2.5. Each block becomes a task. Dependence arcs become successor pointers. Outer serial compound functions do not become tasks, but the innermost level of nested compound functions do become tasks for scheduling. The enclosing compound functions instead cause a duplication as the serial loops are unrolled.

Now as was previously mentioned, COUNT is a very short program. Figure 2.5 makes clear that in this case, a scheduler has very little to decide. (It must still decide whether or not to fold the two parallel tasks). Larger programs obviously are much more complex, containing many separate paths from which to chose, but would be too complex for the illustrative purpose of this section.

## 2.3. The Scheduling Simulation

Once the DAGs from one or more programs have been generated, they are ready for scheduling. A run–time scheduling simulator was written which reads in the DAGs representing the programs, and schedules them using a variety of different scheduling algorithms. Since the specific manner in which the simulator operates to a large extent defines the model for the data produced, its algorithm will be discussed below. Input parameters to the software as well as the information collected and returned by the simulation are described in detail in the Appendix.

The user must initially specify such things as the scheduling technique which is to be applied to the DAG, the machine size, folding requirements, etc. Either a single scheduling algorithm may be used, or the complete series of algorithms will be run on the same DAG and machine. In collecting data for this thesis, most DAGs were run with all possible scheduling techniques, using 8, 32, 64, 128, and 1024 processors, and with folding requirements of 25% and 100%.

The simulator first begins by reading in nodes of the graph. Each node can specify an arrival time. A "wake up call" is entered into a time–sorted event queue

with that arrival time. When the system clock reaches that value the node is read in, dependence links are generated, and any nodes this new arrival may affect dynamically (described in the next chapter) are visited.

A "starting queue" is maintained of all nodes which have already arrived in the system, have not yet begun execution, and do not have any predecessors with dependence links to this node still remaining in the system. All nodes on the starting queue are candidates for immediate execution.

The simulator sorts the starting queue according to the current scheduling technique. (Folding is taken into account, but is not actually done at this time). The highest priority task as determined by the scheduler is placed at the head of the queue.

Following that, the simulator assigns tasks to processors, in order, from the starting queue. As each task is assigned for execution it is removed from the starting queue, any folding is performed, the number of processors currently available is reduced by the current task's processor requirements, and a "wake up call" is entered into the time–sorted event queue telling the simulator when the task will complete. This process continues until either the starting queue is empty, or else no more tasks can fit on the processors.

When no more nodes can be assigned for execution from the starting queue, the simulator checks the time–sorted event queue for actions which it can take at the current system time. (If there are none at the current time, then the system clock is reset to the earliest time the next event is to occur, and flow continues). This would

include reading in new node arrivals and entering them into the graph (and possibly the starting queue) or stopping execution of tasks which are due to complete at the current time.

If a task has completed, the simulator must release the processors the node reserved, and visit all successors of that node with dependence links. If any of that node's successors have no other predecessor dependence links, then that successor will be entered into the starting queue. Finally, the node is removed from the system.

When all of the events which can occur at the current system time have been processed, the simulator loops back to check for task initiation again and the entire process is repeated.

# CHAPTER 3

# PROGRAMS AND ALGORITHMS

## 3.1. Data Programs

This thesis attempts to determine how well a variety of different scheduling algorithms perform on real benchmark programs executed on a typical multiprocessor. Unlike abstract theoretical models, the characteristics of functioning programs taken from the field are not necessarily known in advance, and can only be determined from a compiler such as Parafrase. And yet this area is of crucial importance if software designers intend to efficiently exploit the parallel hardware on which they run.

A representative selection of benchmark programs was collected for detailed study in this work from well known sources such as Eispack [SBDG76] and Linpack [DBMS79]. The FORTRAN programs analyzed by Parafrase for use in the run–time scheduling simulations are shown in Table 3.1. They represent a mix of different applications likely to be run on a parallel system, and both very short and very long programs have been included.

A graphical representation of the DAG for one program in Table 3.1, COUNT, has already been shown in Figure 2.5. COUNT, however, is a very short and simple program. More elaborate examples are shown for illustrative purposes in Figures 3.1 and 3.2. The two programs shown are of moderate size (in terms of complexity).

| Table 3.1. FORTRAN Data Programs. | | |
|---|---|---|
| Program | Source | Purpose |
| CGECO | Linpack | Factor a complex matrix by Gaussian elimination |
| CHEBY | ACM | Simultaneous Chebyshev analysis of nf functions |
| COUNT | ACM | Counts number of partitions of an integer |
| FIGI | Eispack | Reduce tridiagonal matrix to symmetric tridiagonal with same eigenvalues |
| FIGI2 | Eispack | Reduce tridiagonal matrix to symmetric tridiagonal |
| HTRIBK | Eispack | Eigenvectors of complex Hermitian matrix |
| KERNEL | Lawrence Berkley Lab | Newton's search for inversion |
| THREEDH | Fishpack | Three dimensional Helmohltz solver using FFT (Separable elliptic partial differential equations) |

The most extensive programs would obviously take several pages each to diagram. (Note that program complexity, or the number of nodes and arcs in a DAG, may be unrelated to that program's execution time or processor requests).

A few of the symbols in Figures 3.1 and 3.2 need to be discussed. The first symbol is the vertical bracket, which also appeared around $N_C$ and $N_D$ in Figure 2.5. As was briefly mentioned earlier, this represents an outer nested serial loop. To generate the DAG for the run–time scheduler, the serial loop must be unrolled by replicating the nodes and arcs within that loop sequentially. The serial loop count is shown to the left of the vertical bracket. Also, between the $i^{th}$ and the $i+1^{th}$ iteration of the loop, dependence arcs must be generated between the terminal nodes of one iteration of the loop and the starting nodes of the next iteration.

A new symbol introduced in Figures 3.1 and 3.2 is the horizontal bracket, such as is seen around $N_P$ in Figure 3.2. The horizontal bracket represents an outer

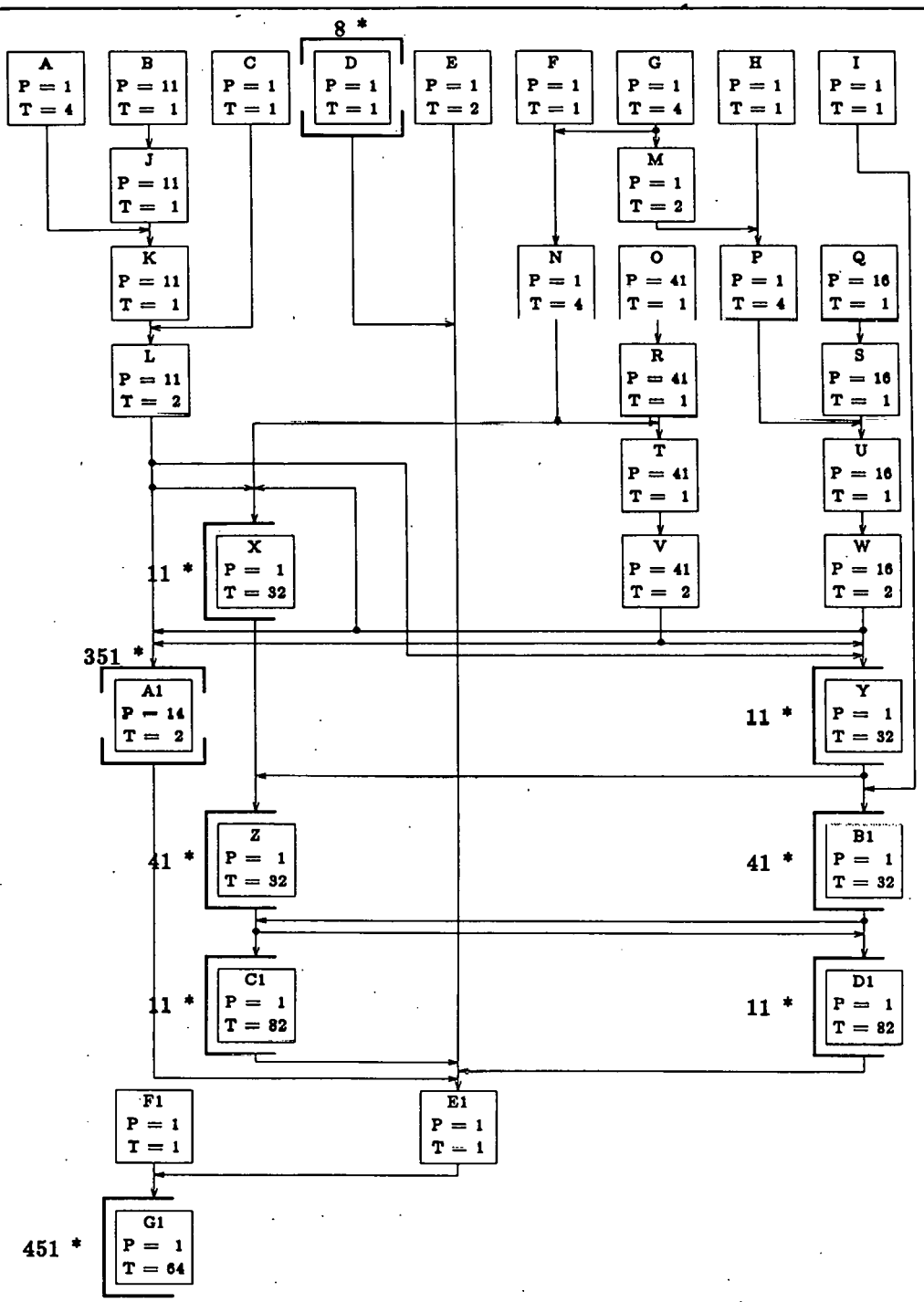



Figure 3.1. Program THREEDH.

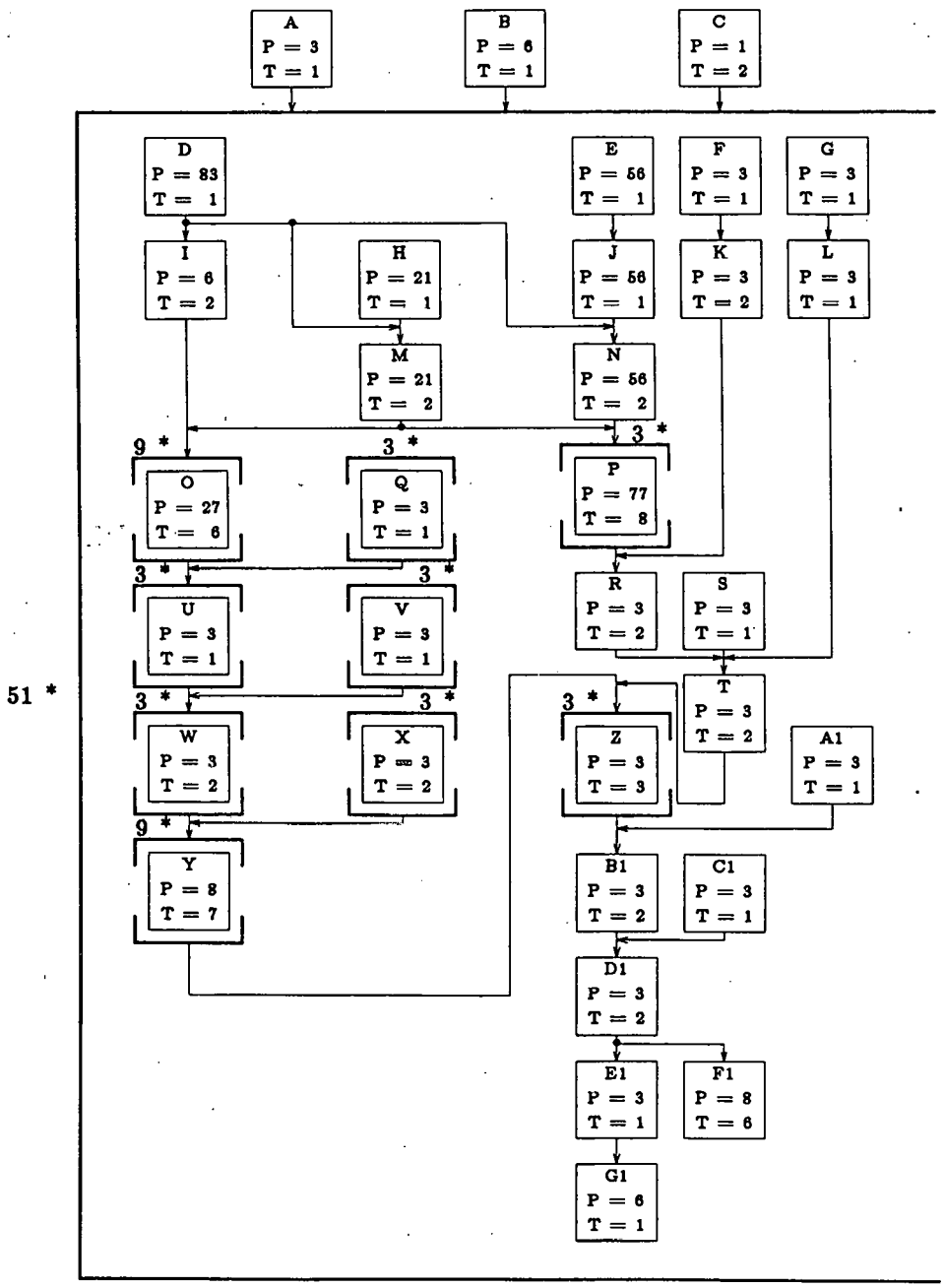



Figure 3.2. Program KERNEL.

nested parallel loop. Nodes within parallel brackets are replicated in parallel, i.e., no dependence arcs between each copy of the node is generated. But of course any dependence arcs entering or leaving the horizontal brackets must be passed on to each copy of the node generated for the DAG. The outer nested parallel loop count is displayed above the upper horizontal bracket.

Horizontal and vertical *brackets* should not be confused with parallel and serial *nodes.* For example $N_D$ in Figure 3.1 is a serial node since the number of processors requested is 1. However, 8 copies of $N_D$ must be generated due to the enclosing parallel loop. Each of the 8 copies may be executed in any order, or at the same time. Furthermore, all 8 copies of $N_D$ must have completed before $N_{E1}$ can be scheduled for execution. Likewise, $N_B$ in Figure 3.1 is a parallel node, even though it has no enclosing parallel brackets, since the number of processors requested is 11. Only one copy of $N_B$ exists, however.

In a similar fashion $N_K$ in Figure 3.2 is a parallel node while $N_C$ is not, even though $N_C$ is external to the vertical (serial) bracket while $N_K$ is nested within. The point is that the presence of vertical and horizontal brackets is irrelevant to whether or not any particular node is serial or parallel.

Use of these symbols allows very large DAGs to be displayed in the simple fashion shown in Figures 3.1 and 3.2. For example, Figure 3.2 displays only 33 boxes. However, the DAG the run–time scheduler will see at execution time contains 3063 nodes and 11,237 arcs! Obviously, scheduling such a program is a formidable task for any heuristic which must operate in real time.

Each new DAG in Table 3.1 has its own interesting features and is of course unique. Just to demonstrate a few types of environments a run–time scheduler may encounter, consider Figures 3.1 and 3.2 more closely. In Figure 3.1, program THREEDH, all of the parallel and serial replication, i.e., horizontal and vertical brackets, is around *single* nodes. Also, a very wide variety of values within each node is present in program THREEDH. For example, $N_{D1}$ requests 1 processor for 82 units of time, while $N_R$ requests 41 processors for only a single unit of time.

Another interesting feature of program THREEDH is nodes such as $N_{A1}$ and $N_{G1}$. All 351 copies of $N_{A1}$ request 14 processors, for a total of 4914 processors which, in theory, could be utilized concurrently. On the other hand, all 451 copies of $N_{G1}$ must be run sequentially, and only need a single processor. THREEDH contains several relatively short and fat or long and skinny nodes. So what size machine should THREEDH be run on and how many processors should be allocated for its execution? Should all 351 copies of $N_{A1}$ be executed simultaneously, or would that lead to intolerable hardware utilization later on in the program? Obviously, with the rich environment programs such as THREEDH offer, a good run–time scheduler must be capable of adapting to the changing values of the starting nodes as it moves through the DAGs executing.

The most distinguishing feature in Figure 3.2, program KERNEL, is the large outer nested serial loop which encloses most of the program. Unlike THREEDH where the vertical brackets enclosed single nodes only, KERNEL's serial loop encases all but three nodes.

Yet within each iteration of that loop, a run–time scheduler has many factors to consider and from which to chose. Many different types of parallelism exist within the nested serial loop. For example, individual nodes themselves, e.g. $N_D$, exhibit a high degree of parallelism. Multiple control paths exist which a run–time scheduler may have to select from, e.g. $N_E$ and $N_G$. And finally, outer nested parallel loops create a variety of nodes and paths the run–time scheduler must traverse, e.g. the nine copies of $N_O$.

## 3.2. Scheduling Algorithms

Twenty five different algorithms were used for simulation studies of a multiprocessor run–time scheduler on the benchmark data programs listed in Table 3.1. The scheduling algorithms used are listed in Table 3.2. They are defined and discussed in more detail below, along with a rationale for why each of the scheduling techniques was chosen.

### 3.2.1. Optimal and Random

Ideally, the first scheduling algorithm implemented should be the optimal scheduler, which of course would provide a lower bound on the running time of any collection of DAGs scheduled for execution. Unfortunately, as was shown in Section 1.2, the problem is NP–hard and thus there is little hope of solving the problem in polynomial time.

A brute force attack on the problem also seems out of the question. Only the very smallest of the data programs shown in Table 3.1 lend themselves to a solution

Table 3.2. Run Time Scheduling Algorithms.

| # | Scheduling Technique |
|---|---|
| 1 | Random |
| 2 | First in first out |
| 3 | Largest processor request; ties by largest dynamic critical path |
| 4 | Largest dynamic critical path; ties by largest processor request |
| 5 | Largest dynamic critical path; ties by smallest processor request |
| 6 | Largest dynamic critical path; ties by largest execution time |
| 7 | Largest dynamic critical path; ties by smallest execution time |
| 8 | Largest dynamic critical path; ties by largest product of time & processors |
| 9 | Largest dynamic critical path; ties by smallest product of time & processors |
| 10 | Largest processor request |
| 11 | Smallest processor request |
| 12 | Largest execution time |
| 13 | Smallest execution time |
| 14 | Largest product of time and processors |
| 15 | Smallest product of time and processors |
| 16 | Smallest dynamic critical path |
| 17 | Largest dynamic critical path; ties by largest dynamic critical volume |
| 18 | Largest dynamic critical volume; ties by largest dynamic critical path |
| 19 | Largest dynamic critical volume; ties by largest processor request |
| 20 | Largest dynamic critical volume; ties by smallest processor request |
| 21 | Largest dynamic critical volume; ties by largest execution time |
| 22 | Largest dynamic critical volume; ties by smallest execution time |
| 23 | Smallest dynamic critical volume |
| 24 | Largest dynamic critical path of the earliest program of the same type of program having the largest overall dynamic critical path |
| 25 | Dynamic critical ratio |

of trying all possible combinations. Moderate to large sized programs are simply too big. For example program CHEBY contains almost four thousand nodes and seventy five thousand arcs! Furthermore, each of the individual nodes may be folded at processor allocation time, depending on what other nodes are currently executing in the system. This interaction between the nodes selected for execution which overlap in

the same time slice increases the complexity of the problem dramatically. Also, the programs are often run in a multiprogramming environment, making the size of the data much larger still. Additionally, the DAGs may be dynamic, i.e., the run–time scheduler is not allowed to see the entire graph all at once in those cases, and must make its decisions based only upon partial knowledge of a time–variable graph. And finally, the problems must be solved repeatedly for various machine sizes, which may affect the best execution order of the nodes in any particular DAG. For all of these reasons, an optimal solution simply isn't feasible.

Since an optimal solution to the general problem is not possible, the random scheduling algorithm has been chosen as the standard by which all other scheduling techniques will be judged. That is, all nodes on the starting queue will be sorted randomly before one is selected for execution on the processors.

The random scheduler is used as a sort of "worst case scenario" by which other schedulers can be measured for performance. It is not, of course, an upper bound on execution time in a theoretical sense, but as a practical matter any scheduler which cannot outperform a random scheduler (or even does worse) can be judged as "bad". It is therefore felt that this is an appropriate yardstick for comparison (in conjunction with comparisons *between* the other 24 algorithms, of course).

### 3.2.2. Greedy, Generous, and FIFO

Several scheduling algorithms look primarily at the number of processors requested by a task or that task's execution time. There exists a class of schedulers

which has in the past been commonly referred to as "greedy algorithms".

Scheduling algorithm numbers 3, 10, 12, and 14 are examples of greedy algorithms which examine the largest processor request or the largest execution time. One of the main rationales for using greedy algorithms is that large tasks should be allowed to run first. For example, if a starting task has an unusually high processor request, it may have trouble obtaining a sufficient number of free processors on which to run later on during program execution. If, on the other hand, enough processors are currently free to satisfy the task's requirements, a greedy algorithm will initiate the task immediately. In a similar fashion, long running tasks which tie up the processors for extended periods of time are started at once by greedy algorithms, while shorter tasks are pigeonholed where appropriate in the gaps between the bigger tasks.

The opposite of a greedy algorithm is referred to as a "generous algorithm". Scheduler numbers 11, 13, and 15 are examples of generous algorithms which examine the smallest processor request or the smallest execution time. The rationale for generous algorithms is that small short tasks can get on and off the hardware faster than big slow ones can. Thus, a higher percentage of the total nodes in the DAG may be able to run concurrently or complete in a shorter period of time with a generous algorithm than with a greedy one. This tends to help the average turnaround time for each node.

As with all heuristics, cases can be found which cause the algorithms both to succeed and to fail. For example, consider the four short DAGs shown in Figure 3.3.

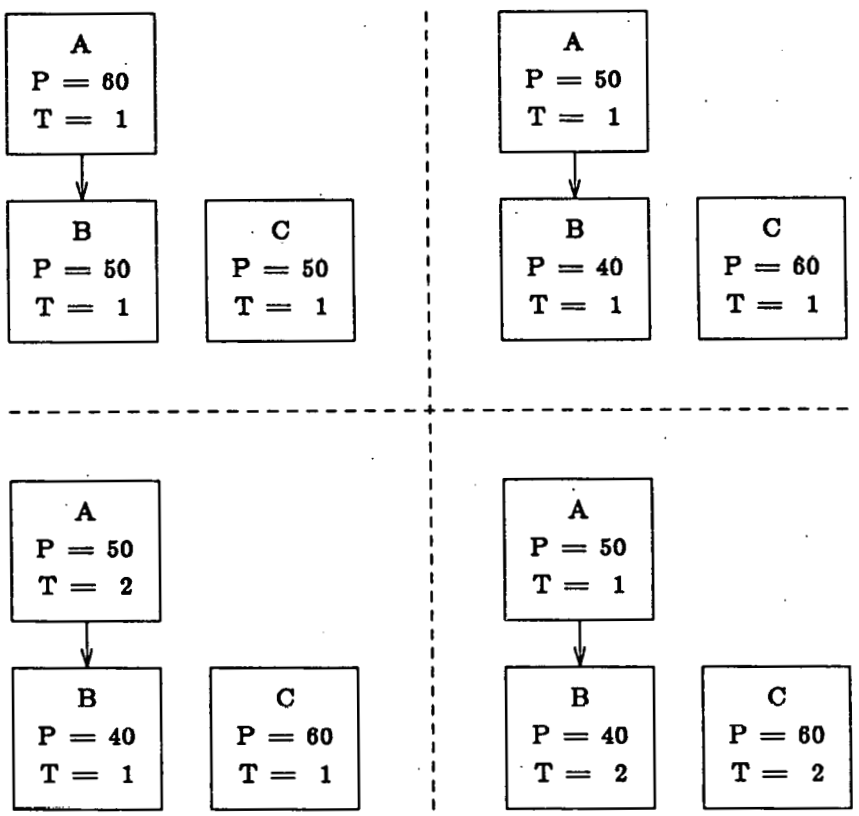



Figure 3.3. Greedy and Generous Comparisons.

Assume a 100 processor machine with no folding.

In the upper left hand corner, a greedy processor algorithm succeeds with a total time of 2, while a generous processor algorithm fails with a total time of 3. On the other hand, a greedy processor algorithm fails with a total time of 3 in the upper right hand corner, while a generous processor algorithm succeeds with a total time of 2. In a similar fashion, a greedy execution time algorithm succeeds in the lower left

hand corner, requiring 3 time units while the generous execution time algorithm needs 4 time units to complete. The opposite holds true in the lower right hand corner, requiring 5 and 3 time units for the greedy and generous execution time algorithms, respectively.

In addition to the greedy and generous algorithms just listed which examine processor requests or execution times as their primary selection criteria, many other algorithms in Table 3.2 use this as a secondary criteria in the event of a tie by their first choice. Furthermore, even when other scheduling techniques are used (for example those discussed in the following two sections) for comparison purposes the primary evaluation criteria has been negated in a greedy/generous fashion, i.e., scheduler numbers 16 and 23.

Scheduling technique number 2 is the standard FIFO, or first in first out algorithm. Its rationale springs from the idea that the tasks which have been waiting in the system for the longest period of time should be served next. The FIFO algorithm tends to minimize turnaround time. FIFO is also used as a subcomponent of scheduler number 24.

### 3.2.3. Dynamic Critical Path

Several of the schedulers shown in Table 3.2 use a form of algorithm known as the *dynamic critical path algorithm.* The dynamic critical path algorithm is listed in Figure 3.4. It is discussed in detail below.

```
subroutine dcpath (node , current_distance)
if node (dcpd) ≥ current_distance
   then return
   else begin
        node (dcpd) = current_distance
        current_distance = current_distance + node (execution_time)
        for all node (predecessor_i) do
             call dcpath (predecessor_i , current_distance)
        return
        end
end dcpath
```

Figure 3.4. Dynamic Critical Path Algorithm.

In order for a scheduler to make use of the dynamic critical path algorithm, each of the nodes in the DAG must be modified to contain not only the node's ID, processor request, and execution time, but also a tag known as the *dynamic critical path distance* (dcpd in Figure 3.4). The dynamic critical path distance for each node is the largest sum of the execution times for all nodes along one of that node's successor paths. That is, assume $N_\alpha$ is some node in a program DAG. Let $T_{N_\beta}$ be the execution time for node $N_\beta$. Then let $S$ be the sequence of nodes ($N_1$, $N_2$, ... $N_t$) such that $N_\alpha > N_1$, $N_{k-1} > N_k$, and $N_t$ is a terminal node, i.e., $N_t$ has no successors. Assume there are $\sigma$ such sets of $S$. Then the dynamic critical path distance for $N_\alpha$ is 0 if $N_\alpha$ is a terminal node, or $\max_{1 \le i \le \sigma} \left\{ \sum_{N_j \epsilon S_i} T_{N_j} \right\}$.

The dynamic critical path algorithm is similar to standard critical path algorithms in that the critical path for a graph is computed for each node. And obviously in this application, the scheduler selects the starting node with the highest critical path value as the next one to schedule for execution. One of the modifications used in this model, however, is that the DAGs may be dynamically modified at run time to add new nodes and arcs into the graph. Clearly, this has the potential for altering the critical path to any of the nodes in the DAG.

Since the DAGs are not static, the dynamic critical path algorithm shown in Figure 3.4 dynamically recalculates the critical path distances at run time whenever the graph is modified. That is, the dynamic critical path algorithm is called once each time the operating system links a new node into the pool of available code segments. This is essential if the scheduler is to be able to correctly determine the critical path in a DAG constantly under revision.

The dynamic critical path algorithm seems to strike at the heart of the scheduling problem. That is, if several different options are presented to a scheduler, it seems clear that a scheduler should begin work immediately on the path that will take the longest, and hope that it can overlap shorter paths "on the fly" as it goes. For example, there is no reason to initiate $N_{F1}$ in program THREEDH until well into program execution, even though it is an original starting node and may have been in the system a long time. Looking at Figure 3.1, it becomes obvious that other work is much more important, and $N_{F1}$ can easily be run concurrently with other nodes any time a free processor becomes available.

Critical path studies are well known in the operations research area. And the dynamic critical path algorithm does in fact correctly calculate the true critical path for any DAG as that DAG is constructed (or modified). However, unlike the results presented in work such as [Hu82], there is no guarantee that selecting a node with the highest dynamic critical path distance value will return optimal results, for the reasons cited in Section 1.2. Nevertheless, on average it should be expected that the dynamic critical path algorithm will return "close" to optimal results.

Figure 3.5 shows the dynamic critical path algorithm in action. In the upper segment, a DAG of three nodes already exists in the system. The dynamic critical path distance value for each of the nodes is marked with the tag of D. Since $N_A$ has two successors, each having an execution time of 100, $N_A$'s dynamic critical path value is 100.

In the lower left hand segment, $N_D$ has been added, such that $N_B > N_D$. The dynamic critical path algorithm must therefore be called. $N_D$'s only predecessor is $N_B$. Its old dynamic critical path value was 0, and so it is now reset to 100, $N_B$'s new distance to the base of the tree. Likewise, $N_A$ is also shifted up by 100, giving it a new dynamic critical path value of 200. (If $N_A$ had any predecessors, the process would continue up the chain). This new value for $N_A$ thus represents the largest sum of the execution times for all nodes ($N_B$ and $N_D$) along one of $N_A$'s two successor paths ($N_A > N_B > N_D$ and $N_A > N_C$). Since $N_C$ is not along any of the dependence paths of $N_D$'s predecessors, it is not visited.

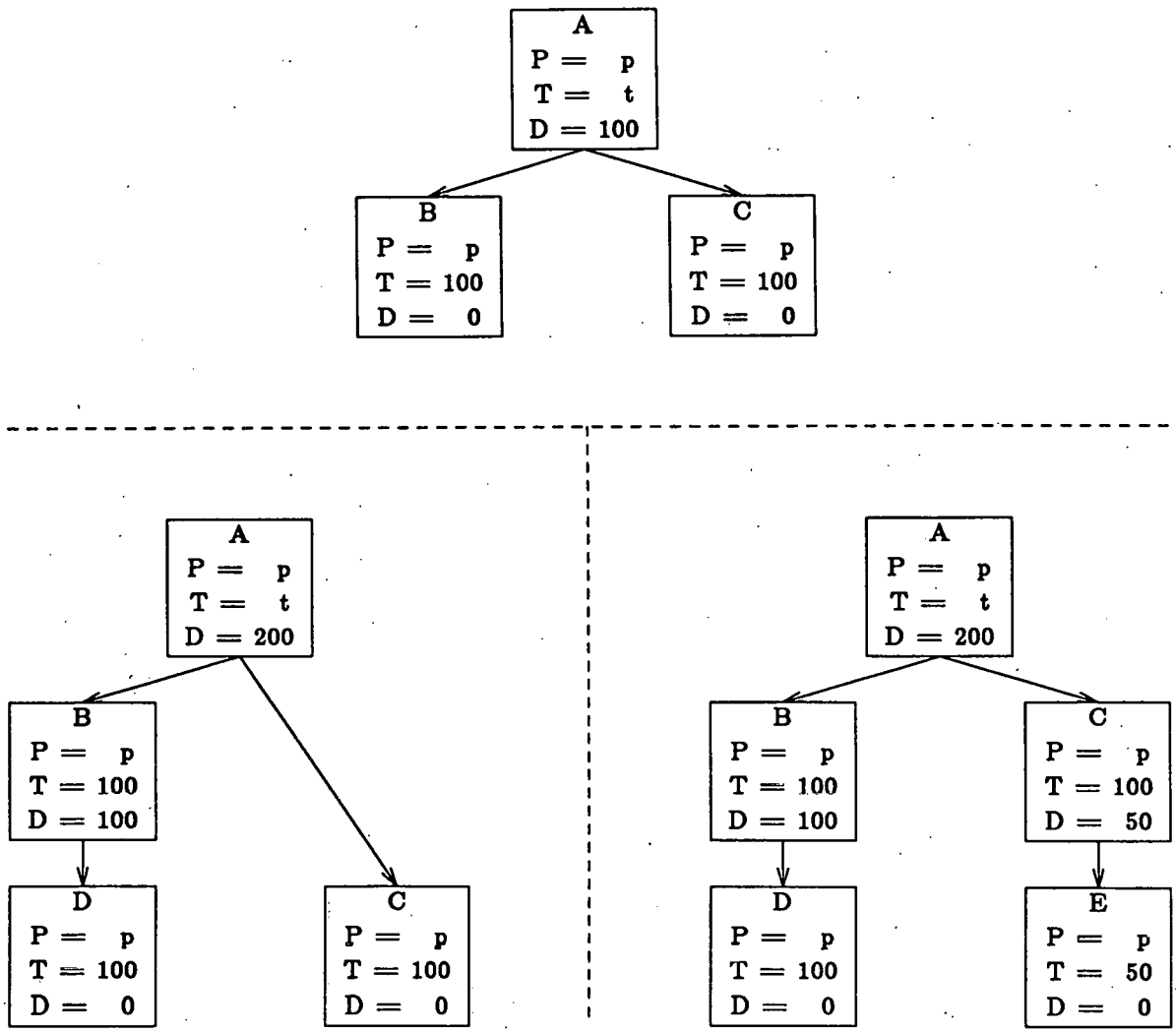



Figure 3.5. Insertion of New Nodes with Tag Modifications.

In the lower right hand segment, $N_E$ has been added, such that $N_C > N_E$. $N_C$ has been shifted up by 50. $N_B$ and $N_D$ have not been visited. Note that the dynamic critical path algorithm terminated on $N_A$, without shifting it up, since its

dynamic critical path value was larger than the path represented through $N_C$. (If $N_A$ had any predecessors, the process would therefore *not* continue up the chain).

Figure 3.6 shows a simple program DAG where the dynamic critical path algorithm returns an optimal result of 300 total time units (assuming a machine of 100 processors without folding). Note that unless a node from the critical path of $N_C > N_D > N_E$ is selected at each opportunity, a suboptimal result of 400 is inevitable. Greedy and generous algorithms such as those discussed in the previous section have no basis on which to make a decision in this instance, and are likely to return a result of 400, as would a random scheduler.

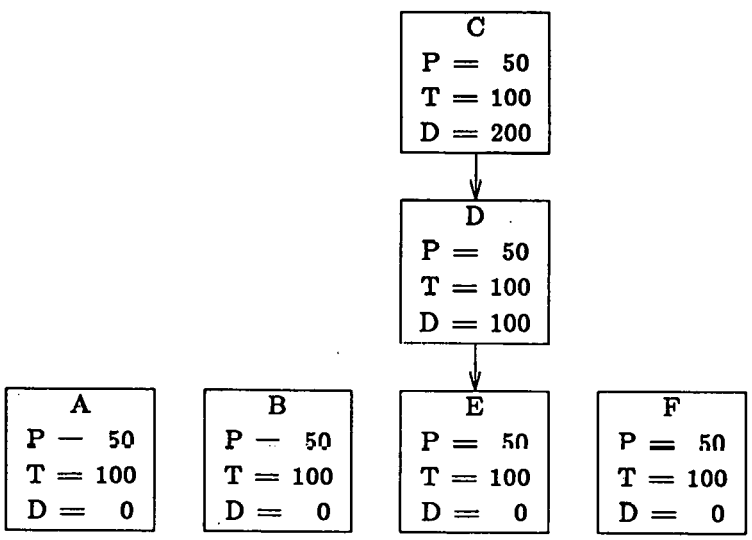



Figure 3.6. Program DAG.

One of the potential pitfalls with any run–time scheduler is the problem of overhead. A good run–time scheduler cannot be too complex, otherwise it runs the risk of becoming a potential bottleneck in the system. If that were to happen, then even a random scheduler would be an acceptable solution, rather than slowing down the system with scheduling overhead.

Fortunately, the dynamic critical path algorithm has three good points which can be made in its favor which help to keep overhead to a minimum. First, the algorithm only visits nodes along the predecessor paths of the new node being entered into the system. If a node does not have this new node as a (direct or indirect) successor, it will not be examined. Additionally, the algorithm terminates along a path as soon as a node is reached who's dynamic critical path distance exceeds the current critical path distance (current_distance in Figure 3.4). Thus, only a small subset of the total DAG is likely to be visited on average.

Second, it may be that some particular program is known to be static throughout its life in the system. In such cases, the critical path distance tags can be precalculated by the compiler, eliminating the need for the costliest portion of this scheduling technique to be performed at run time. Furthermore, even if a program DAG is to be dynamic, the compiler can still generate the tags for those portions of the program which are to be entered into the system at the same time.

Third, this scheduling technique allows for an interesting division of labor. It may be broken down into two disjoint responsibilities: the *dispatcher* and the *scheduler*. Both programs may be run in parallel.

The scheduler's responsibility is to run the dynamic critical path algorithm listed in Figure 3.4 whenever a new node is linked into a DAG. As it backtracks through the DAG (dependence arcs are implemented by means of a doubly linked list) updating the dynamic critical path distances, the dispatcher can be busy selecting the node with the largest dynamic critical path value from the starting queue which will fit (subject to folding considerations) on the currently available processors, and then allocating those processors for that node.

For several reasons, it does not matter if the scheduler is partially through a DAG modifying the tags when the dispatcher examines the starting queue's dynamic critical path values. First, the scheduler only modifies each node's critical path tags, not any of the dependence arcs or the starting queue itself. Thus, even though the DAG is being written to at the same time another program is reading it, the scheduler cannot violate the program's semantics. (The only danger is involved when the scheduler tries to write to a tag in the starting queue, from which the dispatcher may be busy removing nodes. In that case, the operating system must place a lock around a single node only for either the scheduler or for the dispatcher.)

Second, the worst thing that can happen is that the dispatcher selects the wrong node for execution. This can occur if a new node's entry causes a starting node to receive the largest dynamic critical path value in the DAG, but the scheduler has not completed its work before the dispatcher selects the next node for execution. This, however, is a very minor problem. This dispatcher will simply select the *current* highest tag value, which is probably a good choice anyway, since it is the largest

dynamic critical path distance in the particular instance of the DAG *before* the current node arrived. After all, whenever the possibility of dynamic graphs are allowed, any scheduling technique must resolve itself to making decisions based upon incomplete knowledge, i.e., it only has a local rather than global view of what's "good".

Since the (relatively costly) execution of the scheduler may be overlapped with that of the dispatcher (and of course with task execution) at least some of the overhead in implementing a dynamic critical path algorithm can be eliminated. When combined with the facts that the scheduler may only need to visit a subset of the graph, and that compiler assist is possible in generating the nodes' tags, it is felt that the dynamic critical path algorithm is a good candidate for study and possible implementation on a multiprocessor.

### 3.2.4. Dynamic Critical Volume

Several of the schedulers shown in Table 3.2 use a form of algorithm known as the *dynamic critical volume algorithm.* The dynamic critical volume algorithm is listed in Figure 3.7. It is discussed in detail below.

Upon close inspection of Figure 3.7, it becomes clear that the dynamic critical volume algorithm is very close to that of the dynamic critical path algorithm listed in Figure 3.4. In fact, the only difference between the two scheduling techniques is the manner in which the critical path is defined. The dynamic critical volume algorithm defines the *dynamic critical volume* for each node (dcvl in Figure 3.7) to be the larg-

```
subroutine dcvolume (node , current_vol)
if node (dcvl) ≥ current_vol
   then return
   else begin
        node (dcvl) = current_vol
        current_vol = current_vol + node (execution_time) * node (processor_req)
        for all node (predecessor_i) do
             call dcvolume (predecessor_i , current_vol)
        return
        end
end dcvolume
```

Figure 3.7. Dynamic Critical Volume Algorithm.

est sum of the product of the execution times with the processor requests for all nodes along one of that node's successor paths. That is, if $P_{N_\beta}$ is the processor request for node $N_\beta$, then the dynamic critical volume for $N_\alpha$ is $\max_{1 \le i \le \sigma} \left\{ \sum_{N_j \epsilon S_i} T_{N_j} P_{N_j} \right\}$.

The rationale for this modification is that the dynamic critical path algorithm only looks at the sum of the execution times along some path of the DAG. Such a sum, however, may not represent a true reflection of the work involved in executing that set of nodes on the processors. A more accurate measure of the work might be the space–time products of the nodes within that set.

Furthermore, in this model a one–way mapping from processors into execution time is possible, as was discussed in Section 2.2. And if the 25 percent folding rule is permitted, then the dynamic critical path distance for each node potentially may be off by a factor of 4 from the real time needed to execute that path. The dynamic critical volume for each node takes this into account.

Despite the differences in the manner in which the critical path is defined, virtually all of the characteristics of the dynamic critical path algorithm hold true for the dynamic critical volume algorithm. Section 3.2.3 covers such characteristics in detail, and thus they are not repeated here.

### 3.2.5. Throughput Tradeoffs

Up to this point, all of the scheduling techniques have been directed for the most part towards maximizing throughput and processor utilization. As any operating system designer knows, however, throughput is not the only important criteria.

From the users' point of view, program turnaround time may be as important, and perhaps even more so, than machine throughput. Most users are willing to trade some degradation in machine performance (as long as it is not too big) in order to receive quick response time. A good operating system tries to balance these two competing interests.

Scheduling technique number 24 in Table 3.2 tries to make such a balance. This scheduler picks as the next node to execute the largest dynamic critical path of the earliest program of the same type of program having the largest overall dynamic crit-

ical path. That is, it attempts to combine the dynamic critical path algorithm, which tends to maximize throughput, with the FIFO algorithm, which tends to minimize program turnaround time.

Stated another way, the algorithm works as follows. First, the node with the largest overall critical path value is located. Next, the program type that node belongs to is identified. Now, the FIFO portion swings into action. The scheduler determines which program of that same type has been in the system for the longest period of time (this, of course, assumes a multiprogramming environment). Once that program has been determined, the largest critical path node belonging to that program becomes the next candidate for execution. Thus, this scheduler always selects a (relatively) large critical path, which is good for throughput, but runs older jobs first, which is good for turnaround time.

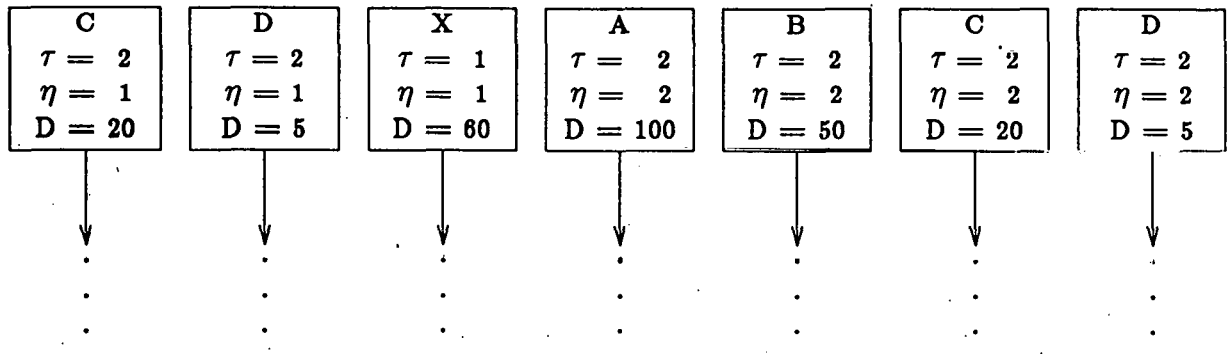



Figure 3.8. Multiprogramming DAGs.

For example, consider Figure 3.8, where $\tau$ represents the program type, and $\eta$ represents the program number. Two different types of programs are shown. Also, two copies of program $\tau_2$ are present; the first one has already had at least two of its nodes serviced, i.e., $N_A$ and $N_B$ (assume it has been in the system for a longer period of time).

Scheduler number 24 then proceeds in the following sequence. Since $N_A$ of $\tau_2\ \eta_2$ has the largest overall value for D, further search is restricted to only $\tau_2$. Next, $\tau_2\ \eta_1$ is selected, since it arrived earlier than $\tau_2\ \eta_2$. And finally, $N_C$ within $\tau_2\ \eta_1$ is scheduled for execution, since it has the largest D value in that program type and number. (The *next* node selected would be $N_D$ in $\tau_2\ \eta_1$ for similar reasons. $N_A$ in $\tau_2\ \eta_2$ follows the completion of $\tau_2\ \eta_1$, and then $N_X$ in $\tau_1\ \eta_1$ is begun before $N_C$ or $N_D$ in $\tau_2\ \eta_2$.)

### 3.2.6. Dynamic Critical Ratio

Unfortunately, scheduling technique number 24 potentially has a serious drawback. That algorithm requires that each node carry a program number tag, along with a program type tag. Although the program number tag could automatically be generated by the operating system as a unique ID, the presence of program type tags requires the users to categorize their programs into a disjoint set of bins. Besides wasting the additional space needed to hold the tags, such a requirement may be unrealistic in a practical environment.

Ideally, a good scheduler should make its decisions based only upon the *characteristics* of the DAGs in the system. The *dynamic critical ratio algorithm* is another attempt to balance the often competing interests of throughput and turnaround time, while at the same time avoiding the drawback of scheduler number 24.

The dynamic critical ratio algorithm is very simple. This scheduler first determines the node with the largest overall critical path. That node is scheduled for execution unless one of two things happen. Either a node can be found with a critical path shorter than the largest node by some *critical ratio*, or a node can be found with a critical path shorter than the *previous* node initiated on the processors by the critical ratio. If such a short node can be found, then it is given a higher priority than the largest critical path node.

Thus, a two node "working set" is maintained for the purposes of evaluating against a critical ratio. This allows the algorithm to adapt, to some extent, to a changing mix of jobs and average critical path values. Note that no requirement is needed for the nodes to carry along program number or type tags.

The rationale for the dynamic critical ratio algorithm is that for machine throughput, big programs with large critical paths should receive a high priority. However, once a job is "close" to finishing, then delaying it further will gain nothing except to drive up the average turnaround time. So in those cases, the jobs with little left to do are quickly flushed out of the system before control returns to the longer programs.

For example, consider the example in Figure 3.8 again. Assume the critical ratio value is 4. Then in this example, since the largest value of D for any of the starting nodes is 100, all nodes with critical path values of less than 25 are initiated first, i.e., all copies of $N_C$, $N_D$, and their successors. Following that, $N_A$ is initiated (this may occur sooner, if sufficient processors are available but $N_C$ and $N_D$'s successors are blocked by dependencies from executing predecessors). The final initiation sequence then continues in decreasing values of D, i.e., $N_X$ then $N_B$.

In some ways, this is analogous to the way many printer queues are set up. That is, the shorter the job, the more important good turnaround time is likely to be to the user. Thus, very short jobs are placed at the head of the queue, even if they arrived later, while medium and longer jobs are serviced normally. (The analogy is not perfect, of course, e.g., print jobs are continuous, while DAGs are composed of smaller subgraphs, which both prevent top level nodes from entering the starting queue if dependencies exist from currently executing nodes, and also must release their processors once the nodes have completed).

# CHAPTER 4

# SCHEDULING SIMULATIONS

## 4.1. Monoprogramming

This section will investigate the effects run-time schedulers have on monoprogramming systems, i.e., deciding between the different control paths of a single job at a time. There are many reasons why a scheduler may only have one job on which to work, e.g., some programs may have such a high priority that no interference is permitted, others may be simply too big to fit on the machine while other jobs are running, certain applications such as operating systems development or real time programs require a dedicated environment, different portions of a larger multiprocessor may be partitioned distinctly between unique jobs, etc.

The object in a monoprogramming system, clearly, is to minimize the total execution time, at the expense of all other parameters. This job cannot affect the performance of any other jobs, since obviously none are present. Furthermore, given a fixed number of processors, there is no reason why all of them should not be utilized by the program if it helps to minimize the execution time. Smaller total execution times directly translate to increased throughput.

As was stated in Section 3.2.1, the random scheduler will be used to compare the relative performances of the other run-time schedulers. The random scheduler is fast, and represents a practical worst case that any good scheduler must be prepared

to beat.

The following tables show the effects the various schedulers have on the programs in a monoprogramming system. An 8 processor machine with 25% folding was used. The SCH columns represent the scheduler numbers (refer to Table 3.2 for the meanings) and the TEX columns represent the total execution times needed to complete the program DAGs on that machine.

Table 4.1. Program CGECO, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 21252 | 2 | 20554 | 3 | 23314 | 4 | 20114 | 5 | 21874 |
| 6 | 20114 | 7 | 21874 | 8 | 20114 | 9 | 21874 | 10 | 22274 |
| 11 | 20514 | 12 | 22274 | 13 | 20514 | 14 | 22274 | 15 | 20514 |
| 16 | 21194 | 17 | 20114 | 18 | 20114 | 19 | 20114 | 20 | 20114 |
| 21 | 20114 | 22 | 20114 | 23 | 20914 | 24 | 21874 | 25 | 21874 |

Table 4.2. Program CHEBY, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 26090 | 2 | 26030 | 3 | 25909 | 4 | 25910 | 5 | 26030 |
| 6 | 25910 | 7 | 26030 | 8 | 25910 | 9 | 26030 | 10 | 25909 |
| 11 | 26104 | 12 | 25909 | 13 | 26124 | 14 | 25909 | 15 | 26124 |
| 16 | 26030 | 17 | 25910 | 18 | 25910 | 19 | 25910 | 20 | 25910 |
| 21 | 25910 | 22 | 25910 | 23 | 26030 | 24 | 26030 | 25 | 26030 |

Table 4.3. Program COUNT, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 3290 | 2 | 3290 | 3 | 3290 | 4 | 3290 | 5 | 3290 |
| 6 | 3290 | 7 | 3290 | 8 | 3290 | 9 | 3290 | 10 | 3290 |
| 11 | 3290 | 12 | 3290 | 13 | 3290 | 14 | 3290 | 15 | 3290 |
| 16 | 3290 | 17 | 3290 | 18 | 3290 | 19 | 3290 | 20 | 3290 |
| 21 | 3290 | 22 | 3290 | 23 | 3290 | 24 | 3290 | 25 | 3290 |

Table 4.4. Program FIGI, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 499 | 2 | 499 | 3 | 499 | 4 | 499 | 5 | 499 |
| 6 | 499 | 7 | 499 | 8 | 499 | 9 | 499 | 10 | 499 |
| 11 | 499 | 12 | 499 | 13 | 499 | 14 | 499 | 15 | 499 |
| 16 | 499 | 17 | 499 | 18 | 499 | 19 | 499 | 20 | 499 |
| 21 | 499 | 22 | 499 | 23 | 499 | 24 | 499 | 25 | 499 |

Table 4.5. Program FIGI2, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4047 | 2 | 5327 | 3 | 3807 | 4 | 5327 | 5 | 5327 |
| 6 | 5327 | 7 | 5327 | 8 | 5327 | 9 | 5327 | 10 | 3807 |
| 11 | 5327 | 12 | 3807 | 13 | 5327 | 14 | 3807 | 15 | 5327 |
| 16 | 3845 | 17 | 5327 | 18 | 3807 | 19 | 3807 | 20 | 3807 |
| 21 | 3807 | 22 | 3807 | 23 | 5327 | 24 | 5327 | 25 | 5327 |

Table 4.6. Program HTRIBK, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 18834 | 2 | 18805 | 3 | 18610 | 4 | 18663 | 5 | 18611 |
| 6 | 18663 | 7 | 18611 | 8 | 18663 | 9 | 18611 | 10 | 18610 |
| 11 | 18816 | 12 | 18610 | 13 | 18870 | 14 | 18610 | 15 | 18848 |
| 16 | 18815 | 17 | 18663 | 18 | 18663 | 19 | 18663 | 20 | 18663 |
| 21 | 18663 | 22 | 18663 | 23 | 18815 | 24 | 18663 | 25 | 18620 |

Table 4.7. Program KERNEL, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 37080 | 2 | 39375 | 3 | 31470 | 4 | 34275 | 5 | 36926 |
| 6 | 34275 | 7 | 36927 | 8 | 34275 | 9 | 36926 | 10 | 30603 |
| 11 | 38864 | 12 | 34326 | 13 | 39477 | 14 | 34326 | 15 | 39476 |
| 16 | 40191 | 17 | 34275 | 18 | 31470 | 19 | 31470 | 20 | 31470 |
| 21 | 31470 | 22 | 31470 | 23 | 40394 | 24 | 33255 | 25 | 33363 |

| Table 4.8. Program THREEDH, 8 Processor Machine, 25% Folding. | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
| 1 | 32673 | 2 | 33385 | 3 | 32899 | 4 | 31683 | 5 | 31493 |
| 6 | 31647 | 7 | 31680 | 8 | 31647 | 9 | 31682 | 10 | 32897 |
| 11 | 31649 | 12 | 31651 | 13 | 32917 | 14 | 31657 | 15 | 32091 |
| 16 | 33146 | 17 | 31647 | 18 | 31635 | 19 | 31635 | 20 | 31635 |
| 21 | 31635 | 22 | 31635 | 23 | 33140 | 24 | 31639 | 25 | 31659 |

Obviously, in most instances, little difference can be detected. Most of the programs differ by less than 5%. Two of the programs have identical results for all of the cases. Only in two of the programs, FIGI2 and KERNEL, were slightly more significant deviations noted.

Why is this the case? Why don't the run–time schedulers affect the total execution times of the program DAGs to a larger degree? Program KERNEL in Table 4.9 begins to explain some of the causes of this phenomenon. The EXE column represents the average number of nodes executing on the processors. The STQ column represents the average number of nodes in the starting queue unable to run due to lack of sufficient processors. The BLK column represents the average number of nodes blocked from execution (and the starting queue) due to predecessors with dependence links still in the system. (Note that a node may be blocked due to predecessors that have not yet begun execution, predecessors which have already started running but have not yet completed, or a combination of both). The EXT column represents the average execution time for each node. The CMP column represents the average completion time for each node (measured from when a node enters the starting queue). The TTT column represents the average turnaround time for each

Table 4.9. KERNEL Parameters, 8 Processor Machine, 25% Folding.

| SCH | EXE | STQ | BLK | EXT | CMP | TTT | PRC |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 6 | 1531 | 24 | 95 | 18626 | 8 |
| 2 | 2 | 14 | 1524 | 25 | 203 | 19790 | 8 |
| 3 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 4 | 2 | 14 | 1518 | 18 | 178 | 17164 | 8 |
| 5 | 2 | 13 | 1519 | 27 | 190 | 18503 | 8 |
| 6 | 2 | 14 | 1518 | 18 | 178 | 17164 | 8 |
| 7 | 2 | 13 | 1519 | 27 | 190 | 18504 | 8 |
| 8 | 2 | 14 | 1518 | 18 | 178 | 17164 | 8 |
| 9 | 2 | 13 | 1519 | 27 | 190 | 18503 | 8 |
| 10 | 1 | 15 | 1529 | 11 | 158 | 15431 | 8 |
| 11 | 2 | 5 | 1524 | 26 | 90 | 19422 | 8 |
| 12 | 1 | 14 | 1529 | 17 | 178 | 17311 | 8 |
| 13 | 2 | 4 | 1518 | 24 | 72 | 19637 | 7 |
| 14 | 1 | 14 | 1529 | 17 | 178 | 17311 | 8 |
| 15 | 2 | 4 | 1518 | 24 | 72 | 19636 | 7 |
| 16 | 2 | 3 | 1519 | 26 | 62 | 19989 | 8 |
| 17 | 2 | 14 | 1518 | 18 | 178 | 17164 | 8 |
| 18 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 19 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 20 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 21 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 22 | 1 | 15 | 1529 | 12 | 162 | 15866 | 8 |
| 23 | 2 | 3 | 1519 | 26 | 62 | 20088 | 8 |
| 24 | 2 | 14 | 1518 | 21 | 172 | 16656 | 8 |
| 25 | 2 | 14 | 1513 | 21 | 172 | 16657 | 8 |

node (task). And finally, the PRC column represents the average number of processors busy executing nodes. A little more detail is provided on these and other parameters in the Appendix.

A quick glance at Table 4.9 reveals that not many tasks are executing at any one time on the processors. In fact, over all 25 schedulers, an average of only 1.6

tasks out of a total of 3063 in program KERNEL are busy executing on the processors.

Potentially, this may be due to one of two reasons. Either a sufficient number of processors are not available to KERNEL, or too many nodes in KERNEL's DAG are blocked because of dependence arcs (i.e., a relatively limited selection of control paths are available to the schedulers from which to initiate nodes).

Examination of the PRC column reveals that usually, all 8 processors (the maximum amount allowed on the machine currently under discussion) are in use. And in fact, on average 11.8 tasks are waiting on the starting queue, a seven to one ratio over the number of tasks executing. So this is certainly one candidate, and insufficient processors cannot be ruled out at this stage.

On the other hand, more interesting statistics can be found in the next few columns yet to be discussed. Table 4.9 shows that over all 25 schedulers, on average throughout the lifetime of KERNEL, 1523 tasks are blocked due to dependence arcs. This is a 929 to one ratio over the number of tasks executing and a 129 to one ratio over the number of tasks on the starting queue waiting for additional processors. Clearly, most of the nodes in the DAG are tied up in this state.

The final three columns also support this view. Averaged across time for all of the schedulers, a task will not be finished for 17596.6 time units after it enters, 150.7 time units after it becomes a starting node, but only 19.5 time units once it begins execution. Once again, it seems as if tasks spend most of their time blocked by dependencies. It can be seen that, for all seven data columns in Table 4.9, the

pattern these global averages set also hold true for each of the 25 rows in the table.

Additional evidence is provided by program CGECO in Table 4.10. CGECO was even less susceptible to differences in the scheduling technique than was KERNEL, as can be seen in Tables 4.1 and 4.7. And, the pattern seen in KERNEL becomes even stronger in CGECO. CGECO has, on average, only 1.2 tasks

Table 4.10. CGECO Parameters, 8 Processor Machine, 25% Folding.

| SCH | EXE | STQ | BLK | EXT | CMP | TTT | PRC |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 557 | 10 | 25 | 7973 | 6 |
| 2 | 1 | 1 | 547 | 17 | 24 | 7538 | 6 |
| 3 | 2 | 0 | 570 | 24 | 29 | 8909 | 6 |
| 4 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 5 | 1 | 1 | 558 | 18 | 26 | 8184 | 6 |
| 6 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 7 | 1 | 1 | 558 | 18 | 26 | 8184 | 6 |
| 8 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 9 | 1 | 1 | 558 | 18 | 26 | 8184 | 6 |
| 10 | 2 | 0 | 562 | 23 | 28 | 8395 | 6 |
| 11 | 1 | 1 | 547 | 17 | 24 | 7518 | 6 |
| 12 | 2 | 0 | 562 | 23 | 28 | 8395 | 6 |
| 13 | 1 | 0 | 547 | 17 | 23 | 7514 | 6 |
| 14 | 2 | 0 | 562 | 23 | 28 | 8395 | 6 |
| 15 | 1 | 0 | 547 | 17 | 23 | 7514 | 6 |
| 16 | 1 | 0 | 553 | 17 | 21 | 7852 | 6 |
| 17 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 18 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 19 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 20 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 21 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 22 | 1 | 1 | 542 | 16 | 26 | 7314 | 6 |
| 23 | 1 | 0 | 550 | 17 | 21 | 7711 | 6 |
| 24 | 1 | 1 | 558 | 18 | 27 | 8184 | 6 |
| 25 | 1 | 1 | 558 | 18 | 27 | 8184 | 6 |

executing, 0.6 tasks on the starting queue, but 550.9 tasks blocked by dependence arcs. The average turnaround time for a task is 7777, the completion time is 25.6, and the execution time is a close 17.9. Furthermore, CGECO only used on average 6 of the 8 processors available to it, unlike KERNEL which used all 8. This lends stronger support to the theory that it is dependence arcs, not insufficient processors, which are the cause of such a limited number of nodes in the program DAGs which are able to execute at any one time.

To be completely sure, however, the effects of changing processors must be studied. All program DAGs were tested on 8, 32, 64, 128, and 1024 processors. Program CHEBY, it turns out, has the largest average processor request. A condensation of the results from program CHEBY is shown in Table 4.11.

As can be seen, adding processors causes the total execution time to drop, as might be predicted. At first, the machines are completely saturated, but eventually not all of their capacity is required. (And eventually, CHEBY moves to an almost perfect space–time product square of 423 processors for about 491 time units). It is important to note, however, that whether the processors are swamped or not, little difference in total execution time is recorded among the various run–time schedulers.

Not shown in Table 4.11 are the EXE, STQ, BLK, EXT, CMP, or TTT columns. However, for all of these cases, on all of the various machine sizes, as well as for all of the other programs tested, the same pattern seen in Tables 4.9 and 4.10 is present. That is, most nodes (often by 2 or 3 orders of magnitude) spend most of their time blocked due to dependence links.

Table 4.11. Program CHEBY on Variable Processors, 25% Folding.

| | 8 Processors | | 32 Processors | | 64 Processors | | 128 Processors | | 1024 Processors | |
|---|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | PRC | TEX | PRC | TEX | PRC | TEX | PRC | TEX | PRC |
| 1 | 26090 | 8 | 10420 | 32 | 3582 | 61 | 1963 | 105 | 491 | 423 |
| 2 | 26030 | 8 | 10414 | 32 | 3607 | 61 | 1966 | 105 | 492 | 423 |
| 3 | 25909 | 8 | 10414 | 32 | 3569 | 62 | 1963 | 105 | 492 | 423 |
| 4 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 5 | 26030 | 8 | 10496 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 6 | 25910 | 8 | 10496 | 32 | 3564 | 62 | 1963 | 105 | 401 | 423 |
| 7 | 26030 | 8 | 10406 | 32 | 3566 | 62 | 1964 | 105 | 491 | 423 |
| 8 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 9 | 26030 | 8 | 10496 | 32 | 3566 | 62 | 1964 | 105 | 491 | 423 |
| 10 | 25909 | 8 | 10414 | 32 | 3569 | 62 | 1963 | 105 | 492 | 423 |
| 11 | 26104 | 8 | 10412 | 32 | 3604 | 61 | 1963 | 105 | 491 | 423 |
| 12 | 25909 | 8 | 10412 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 13 | 26124 | 8 | 10414 | 32 | 3569 | 62 | 1964 | 105 | 492 | 423 |
| 14 | 25909 | 8 | 10414 | 32 | 3569 | 62 | 1963 | 105 | 492 | 423 |
| 15 | 26124 | 8 | 10412 | 32 | 3606 | 61 | 1964 | 105 | 491 | 423 |
| 16 | 26030 | 8 | 10414 | 32 | 3607 | 61 | 1966 | 105 | 492 | 423 |
| 17 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 18 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 19 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 20 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 21 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 22 | 25910 | 8 | 10414 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 23 | 26030 | 8 | 10414 | 32 | 3607 | 61 | 1966 | 105 | 492 | 423 |
| 24 | 26030 | 8 | 10496 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |
| 25 | 26030 | 8 | 10494 | 32 | 3564 | 62 | 1963 | 105 | 491 | 423 |

Thus, the question originally posed concerning Table 4.9 can now be answered. It can consistently be shown that *dependence arcs* constrain to a greater degree the number of nodes in a DAG which can execute than the lack of processors available to

process the *separate control paths* in a single job. And indeed, these two conditions are inversely proportional to each other. For if a single program has a large number of dependence arcs present in it, then it is unlikely to have a large number of independent control paths present. So, although the individual nodes in a DAG may be highly parallel (to the extent that some programs can utilize a *very* large number of concurrent processors on average), it appears that (relatively) limited parallelism exists between the nodes in a single program.

The implications of this discovery are clear. For if the bulk of the nodes in a DAG are inaccessible due to dependence links, then only a tiny fraction of the nodes are available to the starting queue. With a smaller pool of nodes from which to make a decision, the various schedulers have a higher probability of selecting the same node (or all of the nodes, if they will fit) for execution on the processors. This situation severely restricts the options *any* run–time scheduler has to chose from, and thereby diminishes the impact a scheduler can have on the total execution time.

The situation actually grows worse as processors are added. Consider a hypothetical machine with an unlimited number of processors available. On such a hypothetical machine, all conceivable schedulers will act in an identical fashion, i.e., as soon as a task enters the starting queue, it will immediately be dispatched for execution on the processors. If a difference in actions is not possible between the schedulers, then a difference in execution time is also not possible. This is obviously true no matter what scheduler or program DAG is placed in to the system.

Naturally, this implies that in an infinite processor situation, all schedulers act optimally. The execution time for any DAG with any scheduler is then identical to that DAG's largest static critical path, since all the schedulers have to do is to work their way down the critical paths, without having to worry about competition for resources.

Furthermore, the closer a real machine approaches an unlimited processor environment (from the program's point of view) the more likely it is that on average, that program's requirements can be met, and the less likely it is that any run–time scheduler will make much of a difference. Note that this is much more probable in a monoprogramming system than in a multiprogramming system. After all, single jobs compose a subset of a multiprogramming environment. Therefore, monoprogramming is much less likely to stress the (realistically limited) processors, and thus be able to differentiate between schedulers to the same extent.

Returning to the discussion of dependence arcs, if they create such a big problem for schedulers, then what would happen if the program's nodes arrived spread out across time, instead of all at once, as has been the case up to this point? In such a situation, some dependence links may be artificially "broken", i.e., a few nodes may arrive with dependence information, only to find that their predecessors have long since completed and left the system. In such cases, the dependence links obviously never get created. Furthermore, a variety of nodes may arrive in close proximity to each other and immediately be placed in the starting queue together, where under the previous tests, these nodes ordinarily may never have been able to compete

against each other.

Table 4.12 shows one example of a dynamically arriving program, in this case HTRIBK. The 2969 nodes of HTRIBK in this example were spread out to arrive evenly across a time interval of 18834, which was the total execution time for the random scheduler in Table 4.6. (One constraint was placed on the arrival of the nodes, however. No node could arrive *before* all of its predecessors had arrived.) As can be seen from the table, again little difference is observed between the various schedulers. The total execution times between program runs were affected, depending upon the arrival characteristics and the particular program DAG begin tested, but no scheduler ever performed consistently and significantly better on dynamic monoprograms.

Similar results were obtained on single jobs as the folding percentage was changed. Folding affected the overall execution times dramatically, as was discussed in Section 2.2, but it did not have a consistent and significant effect on which scheduler performed best on a single program.

Table 4.12. Program HTRIBK Arriving Dynamically, 8 Processor Machine, 25% Folding.

| SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX | SCH | TEX |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 19467 | 2 | 19411 | 3 | 19240 | 4 | 19265 | 5 | 19248 |
| 6 | 19265 | 7 | 19251 | 8 | 19265 | 9 | 19251 | 10 | 19240 |
| 11 | 19403 | 12 | 19240 | 13 | 19476 | 14 | 19240 | 15 | 19476 |
| 16 | 19414 | 17 | 19265 | 18 | 19259 | 19 | 19259 | 20 | 19259 |
| 21 | 19259 | 22 | 19259 | 23 | 19462 | 24 | 19269 | 25 | 19257 |

The only program for which the run–time scheduler had a very large effect was FIGI2, where a 23% improvement was noted on an 8 processor machine in Table 4.5. FIGI2 is somewhat anomalous, due to the presence of a single "fat" node in the DAG, which completely hogs the processors on a small multiprocessor if it is allowed to run first. As it turns out, it happens to be beneficial to select that path first, and so schedulers which favor fat nodes (such as greedy processor or the largest critical volume algorithms) do better in this particular instance.

Thus, several things have now become apparent. Single jobs only have a (relatively) limited number of independent control paths present. This restricts the number of starting nodes schedulers have to process, increasing the probability that they will act the same. Furthermore, (and perhaps related to that problem) single jobs do not always stress the capacity of a multiprocessor. Thus, all starting nodes are free to be initiated immediately, which makes the decisions of a scheduler irrelevant.

For these reasons (and a few others to be discussed in the next section), run–time schedulers in a monoprogramming system do not (usually) have as significant an effect as one might expect in advance. Therefore, it is probably a wise choice to select a scheduler which is as *fast* as possible, with little overhead placed upon the operating system. Furthermore, as many decisions as possible (such as static critical path tags, if desired) should be moved to the compiler, rather than be evaluated at run time.

## 4.2. Multiprogramming

Multiprogramming offers a completely different environment, and therefore potentially a different set of conclusions, than can be found in a monoprogramming system. For one thing, the presence of multiple jobs offers a larger number of independent control paths, and therefore starting nodes, than on a single user system. Furthermore, the processors are more likely to be stressed due to the addition of multiple jobs. A run–time scheduler's decisions thus become more important than in the previous section. And, as will be seen shortly, additional characteristics of multiprogramming systems greatly affect the choice of run–time schedulers which should be implemented.

Multiprocessors are expensive machines, and for most applications, it is probably not cost effective to run a single job in a dedicated environment. It has been known for quite some time that in virtually all systems, the benefits of time sharing far outweights its detriments. Given that this is true, it is probably a safe assumption to make that any given multiprocessor is likely to be running some form of time and processor sharing operating system. And in order to help efficiently exploit the available parallel hardware, this section attempts to examine that problem as it relates to the run–time scheduler.

On the average, at any one time multiprogramming systems are likely to be running a variety of different jobs. It is very common to find one or a few very large jobs executing, such as a long simulation, along with a much greater number of smaller jobs, such as editors, compilers, or debugging runs of larger programs with

limited data. Unless stated otherwise, this model thus becomes the focus of the following two sections.

And as in the previous section, the random scheduler will be used as a performance index on how good or bad any particular scheduler is said to be, along with comparisons between the various schedulers. For that purpose, the total execution time will again be used to measure throughput, holding all other parameters constant. However, unlike the previous section on monoprogramming, throughput is no longer the only criteria of interest. Turnaround time also must be taken into consideration, as response time is obviously a very important factor to many users when their jobs are executed in a multiprogramming environment. For that purpose, the additional data of individual program turnaround times and execution time spans will be introduced.

Hopefully, some sort of balance can be made between these two often competing interests. Section 4.2.1 looks primarily at the throughput issue, and the first 23 schedulers listed in Table 3.2 are analyzed there. Section 4.2.2 looks primarily at the turnaround time issue, and discusses the final two schedulers separately in that section. However, scheduler number 24 is listed in the tables along with the first 23 schedulers in Section 4.2.1, even though it is not analyzed until the following section. Scheduler number 25 returns a range of values, and thus is not even listed in the tables until it is discussed.

### 4.2.1. Throughput

For a variety of reasons, some of which have already been discussed, run–time schedulers can make more of a difference in the total execution time, and thus throughput, of a multiprogramming system than they do in a monoprogramming system. Table 4.13 is typical of such results.

This particular example assumed a single copy of the large program THREEDH was multiprogrammed with 70 copies of the smaller program COUNT. An 8 processor machine, with 25% folding, and arrival times of 0 were assumed. (The PTT columns represent the program turnaround times, and the ETS columns represent the execution time spans. These columns are defined in the Appendix, and will be discussed in greater detail in the next section.)

Several points are immediately apparent. For instance, significant deviations in the total execution times between the schedulers can be noted. Furthermore, as will be seen in some of the other examples cited, consistent patterns begin to develop concerning which schedulers return the best overall throughput performance. Hopefully, this section will illustrate not only which schedulers perform best, but also the reasons behind that increased performance.

Table 4.13 reveals that the random scheduler (SCH row no. 1) returns a total execution time for the DAGs of 60295. It, however, did not do the worst. The worst scheduler in this instance was the smallest dynamic critical volume scheduler (no. 23), with a total execution time of 66154. This represents a 9.7% *increase* in the total execution time. The other schedulers which did worse than random were, in

Table 4.13. Programs THREEDH and COUNT, 8 Processor Machine, 25% Folding.

| | Global Statistics | | | | | | | | THREEDH | | COUNT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | EXE | STQ | BLK | EXT | CMP | TTT | PRC | PTT | ETS | PTT | ETS |
| 1 | 60295 | 5 | 56 | 1913 | 41 | 543 | 17757 | 5 | 60295 | 60285 | 26659 | 25594 |
| 2 | 65498 | 4 | 41 | 2025 | 40 | 440 | 20238 | 4 | 65498 | 65338 | 18335 | 3317 |
| 3 | 35292 | 8 | 75 | 3203 | 40 | 434 | 17303 | 8 | 34698 | 34698 | 34970 | 34611 |
| 4 | 35850 | 7 | 88 | 3248 | 40 | 510 | 17890 | 8 | 35222 | 35222 | 35523 | 32340 |
| 5 | 35121 | 8 | 82 | 3189 | 40 | 468 | 17184 | 8 | 35121 | 35121 | 34786 | 32420 |
| 6 | 36330 | 7 | 88 | 3290 | 40 | 517 | 18358 | 8 | 36330 | 36330 | 35985 | 32527 |
| 7 | 35896 | 7 | 87 | 3254 | 40 | 509 | 17942 | 8 | 35279 | 35279 | 35578 | 32387 |
| 8 | 36330 | 7 | 88 | 3290 | 40 | 517 | 18358 | 8 | 36330 | 36330 | 35985 | 32527 |
| 9 | 35898 | 7 | 87 | 3254 | 40 | 509 | 17944 | 8 | 35281 | 35281 | 35580 | 32387 |
| 10 | 60419 | 4 | 29 | 1963 | 40 | 301 | 17999 | 5 | 60419 | 59719 | 16905 | 16560 |
| 11 | 64928 | 4 | 42 | 2040 | 40 | 443 | 20214 | 4 | 64928 | 64928 | 18349 | 3317 |
| 12 | 58767 | 5 | 33 | 2007 | 40 | 328 | 17932 | 5 | 58767 | 58767 | 16970 | 16613 |
| 13 | 59609 | 5 | 39 | 1965 | 40 | 386 | 17869 | 5 | 59609 | 59609 | 16732 | 3739 |
| 14 | 58506 | 5 | 29 | 2007 | 40 | 295 | 17824 | 5 | 58506 | 58506 | 16910 | 16553 |
| 15 | 59273 | 5 | 40 | 1975 | 40 | 391 | 17867 | 5 | 59273 | 59273 | 16701 | 3572 |
| 16 | 66068 | 4 | 41 | 2013 | 40 | 443 | 20291 | 4 | 66068 | 65908 | 18335 | 3317 |
| 17 | 36330 | 7 | 88 | 3290 | 40 | 517 | 18358 | 8 | 36330 | 36330 | 35985 | 32527 |
| 18 | 36329 | 7 | 88 | 3289 | 40 | 519 | 18350 | 8 | 36329 | 36320 | 35975 | 32528 |
| 19 | 36329 | 7 | 88 | 3289 | 40 | 519 | 18350 | 8 | 36320 | 36329 | 35975 | 32528 |
| 20 | 36329 | 7 | 88 | 3289 | 40 | 519 | 18350 | 8 | 36329 | 36329 | 35975 | 32528 |
| 21 | 36329 | 7 | 88 | 3289 | 40 | 519 | 18350 | 8 | 36329 | 36329 | 35975 | 32528 |
| 22 | 36329 | 7 | 88 | 3289 | 40 | 519 | 18350 | 8 | 36329 | 36329 | 35975 | 32528 |
| 23 | 66154 | 4 | 38 | 2011 | 40 | 419 | 20274 | 4 | 66154 | 65994 | 18335 | 3317 |
| 24 | 41640 | 6 | 85 | 3132 | 40 | 566 | 20031 | 7 | 39351 | 39351 | 23556 | 3318 |

decreasing order of time: smallest dynamic critical path (no. 16), FIFO (no. 2), smallest processor request (no. 11), and largest processor request (no. 10).

Several of the schedulers were less than 5% faster than the performance of random. They were: largest execution time (no. 12), smallest execution time (no. 13), largest product of time and processors (no. 14), and the smallest product of time and processors (no. 15).

Many of the schedulers showed significant speedups over the random algorithm. These were the entire collection of the dynamic critical path algorithms (nos. 4–9 and 17), the dynamic critical volume algorithms (nos. 18–22), and the largest processor request with ties going to the largest dynamic critical path (no. 3). All of these schedulers showed a speedup over random of approximately 40%, with the fastest scheduler returning a total execution time of 35121, or a 41.8% speedup.

Why were such dramatic results recorded? And why did the dynamic critical path and volume algorithms in particular do so well? The other columns in Table 4.13 begin to answer these questions.

Glancing at the EXT, CMP, and TTT columns, it is clear that tasks still spend most of their time blocked by dependencies from execution. This is to be expected. After all, multiprogramming systems are made up from a collection of individual single jobs, each of which exhibit the same dependence characteristics seen in Section 4.1. Adding more jobs does not break links *within* a job.

The difference, however, occurs in the STQ, relative to the EXE and BLK columns. Now, unlike Section 4.1, the introduction of multiple independent jobs ensures a greater number of nodes in the starting queue. In fact, averaged over all of the dynamic critical path and volume algorithms, 87.3 nodes were idle waiting for

processors to become free. (There were 6700 total nodes in the system). This is a much higher number than was seen in a single user system. Even algorithms which performed poorly had a relatively large pool of nodes in the starting queue.

The assertions made in Section 4.1 are thus justified. Multiple jobs with independent control paths in a multiprogramming system obviously place a greater number of nodes onto the starting queue. More nodes on that queue reduces the probability that different schedulers will select the same node. Furthermore, the additional nodes stress the processors more, making it less likely that the different schedulers perform close to optimal. Therefore, as predicted, the run-time scheduler's decisions become more significant, and therefore more important, with multiprogramming.

As a final note, it can be seen from the PRC column in Table 4.13 that those algorithms which performed best had the highest processor utilization, and in fact completely saturated the machine. This is to be expected. Schedulers which did not do well utilized less processors on average, despite the fact that nodes were available in the starting queue (although on average, the starting queue was shorter for poorly performing schedulers).

What effect does the job mix, in particular the number of small programs, have on these results? Tables 4.14 and 4.15 illustrate the changes observed as the degree of multiprogramming is varied. Table 4.14 contains 50% less, and Table 4.15 50% more, copies of COUNT than was executed in Table 4.13. All other parameters were held constant.

Table 4.14. Programs THREEDH and COUNT, 50% Less Multiprogramming, 8 Processor Machine, 25% Folding.

| | Global Statistics | | | | | | | | THREEDH | | COUNT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | EXE | STQ | BLK | EXT | CMP | TTT | PRC | PTT | ETS | PTT | ETS |
| 1 | 46157 | 3 | 19 | 913 | 40 | 264 | 11261 | 4 | 46157 | 46147 | 12866 | 12459 |
| 2 | 40972 | 4 | 31 | 1039 | 40 | 369 | 11478 | 4 | 40972 | 40812 | 11305 | 3310 |
| 3 | 33549 | 5 | 26 | 1132 | 40 | 264 | 10180 | 5 | 33549 | 33549 | 18694 | 18504 |
| 4 | 31683 | 5 | 40 | 1231 | 40 | 368 | 10550 | 5 | 31683 | 31683 | 19447 | 16477 |
| 5 | 31493 | 5 | 33 | 1167 | 40 | 308 | 9906 | 5 | 31493 | 31493 | 18652 | 16519 |
| 6 | 31647 | 5 | 40 | 1257 | 40 | 369 | 10758 | 5 | 31647 | 31647 | 19708 | 16600 |
| 7 | 31680 | 5 | 39 | 1233 | 40 | 364 | 10561 | 5 | 31680 | 31680 | 19464 | 16506 |
| 8 | 31647 | 5 | 40 | 1257 | 40 | 369 | 10758 | 5 | 31647 | 31647 | 19708 | 16600 |
| 9 | 31682 | 5 | 39 | 1233 | 40 | 364 | 10563 | 5 | 31682 | 31682 | 19466 | 16506 |
| 10 | 46411 | 3 | 15 | 920 | 40 | 220 | 11368 | 4 | 46411 | 46061 | 9393 | 9223 |
| 11 | 48328 | 3 | 21 | 971 | 40 | 305 | 12562 | 3 | 48328 | 48328 | 10036 | 3313 |
| 12 | 45013 | 3 | 19 | 933 | 40 | 258 | 11219 | 4 | 45013 | 45013 | 9426 | 9244 |
| 13 | 46403 | 3 | 17 | 914 | 40 | 241 | 11311 | 4 | 46403 | 46403 | 9352 | 3754 |
| 14 | 44989 | 3 | 17 | 934 | 40 | 245 | 11216 | 4 | 44989 | 44989 | 9455 | 9273 |
| 15 | 45582 | 3 | 19 | 920 | 40 | 261 | 11209 | 4 | 45582 | 45582 | 9339 | 3475 |
| 16 | 49468 | 3 | 21 | 960 | 40 | 305 | 12709 | 3 | 49468 | 49308 | 10024 | 3314 |
| 17 | 31647 | 5 | 40 | 1257 | 40 | 369 | 10758 | 5 | 31647 | 31647 | 19708 | 16600 |
| 18 | 31635 | 5 | 40 | 1257 | 40 | 372 | 10751 | 5 | 31635 | 31635 | 19697 | 16600 |
| 19 | 31635 | 5 | 40 | 1257 | 40 | 372 | 10751 | 5 | 31635 | 31635 | 19697 | 16600 |
| 20 | 31635 | 5 | 40 | 1257 | 40 | 372 | 10751 | 5 | 31635 | 31635 | 19697 | 16600 |
| 21 | 31635 | 5 | 40 | 1257 | 40 | 372 | 10751 | 5 | 31635 | 31635 | 19697 | 16600 |
| 22 | 31635 | 5 | 40 | 1257 | 40 | 372 | 10751 | 5 | 31635 | 31635 | 19697 | 16600 |
| 23 | 49554 | 3 | 17 | 960 | 40 | 263 | 12680 | 3 | 49554 | 49394 | 10025 | 3314 |
| 24 | 31639 | 5 | 43 | 1358 | 40 | 393 | 11615 | 5 | 31639 | 31639 | 14205 | 3316 |

As can be seen from Tables 4.14 and 4.15, the overall patterns seen in Table 4.13 are still true. Obviously, the overall execution times tend to decrease in Table

4.14 and tend to increase in Table 4.15. However, the collection of dynamic critical path and volume algorithms consistently perform very well. Other schedulers show little or no consistent improvement, and in some cases actually do worse than random. The smallest dynamic critical path and volume algorithms do exceptionally poorly.

There is one difference between Tables 4.14, 4.15, and Table 4.13 that should be noted, however. Table 4.13 recorded speedups of around 40%. The latter two tables dropped their speedups to just under 32%.

Why did this happen? The answer should be obvious in the case of Table 4.14. For as the degree of multiprogramming is decreased, the system begins to look more and more like a monoprogramming system. The number of nodes on the starting queue begins to shrink, the processors become less stressed, and the utilization drops. Thus, speedup is definitely recorded, but it is not quite as dramatic.

Table 4.15 is less obvious. Why should increasing the number of copies of COUNT by 50% decrease the speedup by approximately 8%? The answer is that as more copies of the smaller program are added to the system, the *relative* contribution of the larger program to the overall statistics begins to decrease. Eventually, the performance values are made up almost entirely as if the scheduler saw only a collection of identical short jobs, without ever introducing a larger program on to the processors.

This then brings up yet another potential pitfall for run–time schedulers. For even if the processors are stressed such that all nodes can't run immediately, and if a

Table 4.15. Programs THREEDH and COUNT, 50% More Multiprogramming, 8 Processor Machine, 25% Folding.

| | Global Statistics | | | | | | | | THREEDH | | COUNT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | EXE | STQ | BLK | EXT | CMP | TTT | PRC | PTT | ETS | PTT | ETS |
| 1 | 74459 | 5 | 83 | 3100 | 41 | 686 | 24803 | 5 | 74459 | 74449 | 34059 | 30771 |
| 2 | 57572 | 7 | 105 | 4466 | 40 | 673 | 27543 | 7 | 57572 | 57412 | 28580 | 3316 |
| 3 | 49930 | 8 | 105 | 4607 | 40 | 586 | 24621 | 8 | 48936 | 48936 | 49397 | 48857 |
| 4 | 50575 | 8 | 119 | 4669 | 40 | 669 | 25346 | 8 | 49582 | 49582 | 50064 | 46671 |
| 5 | 49843 | 8 | 114 | 4605 | 40 | 635 | 24618 | 8 | 49816 | 49816 | 49312 | 46713 |
| 6 | 51312 | 7 | 120 | 4732 | 40 | 681 | 26054 | 8 | 51296 | 51296 | 50773 | 46965 |
| 7 | 50710 | 8 | 119 | 4674 | 40 | 670 | 25439 | 8 | 49680 | 49680 | 50166 | 46742 |
| 8 | 51312 | 7 | 120 | 4732 | 40 | 681 | 26054 | 8 | 51296 | 51296 | 50773 | 46965 |
| 9 | 50712 | 8 | 119 | 4675 | 40 | 670 | 25441 | 8 | 49682 | 49682 | 50168 | 46742 |
| 10 | 76619 | 5 | 42 | 3096 | 40 | 375 | 25161 | 5 | 76619 | 75569 | 24262 | 23742 |
| 11 | 81528 | 5 | 68 | 3244 | 40 | 620 | 28255 | 5 | 81528 | 81528 | 26654 | 3318 |
| 12 | 75368 | 5 | 46 | 3136 | 40 | 400 | 25096 | 5 | 75368 | 75368 | 24242 | 23710 |
| 13 | 75532 | 5 | 64 | 3107 | 40 | 548 | 25071 | 5 | 75532 | 75532 | 24089 | 3774 |
| 14 | 75357 | 5 | 45 | 3138 | 40 | 392 | 25101 | 5 | 75357 | 75357 | 24276 | 23744 |
| 15 | 76474 | 5 | 63 | 3073 | 40 | 547 | 25103 | 5 | 76474 | 76474 | 23863 | 3434 |
| 16 | 82668 | 5 | 67 | 3205 | 40 | 620 | 28303 | 5 | 82668 | 82508 | 26638 | 3318 |
| 17 | 51312 | 7 | 120 | 4732 | 40 | 681 | 26054 | 8 | 51296 | 51296 | 50773 | 46965 |
| 18 | 51309 | 7 | 120 | 4731 | 40 | 683 | 26046 | 8 | 51293 | 51293 | 50762 | 46965 |
| 19 | 51309 | 7 | 120 | 4731 | 40 | 683 | 26046 | 8 | 51293 | 51293 | 50762 | 46965 |
| 20 | 51309 | 7 | 120 | 4731 | 40 | 683 | 26046 | 8 | 51293 | 51293 | 50762 | 46965 |
| 21 | 51309 | 7 | 120 | 4731 | 40 | 683 | 26046 | 8 | 51293 | 51293 | 50762 | 46965 |
| 22 | 51309 | 7 | 120 | 4731 | 40 | 683 | 26046 | 8 | 51293 | 51293 | 50762 | 46965 |
| 23 | 82754 | 5 | 65 | 3202 | 40 | 603 | 28291 | 5 | 82754 | 82594 | 26638 | 3318 |
| 24 | 58240 | 7 | 118 | 4547 | 40 | 756 | 28428 | 7 | 55951 | 55951 | 32264 | 3319 |

large number of multiple independent control paths are present, then if all of those paths have identical or even close characteristics, the scheduler has no basis upon

which it should make a decision. Every path looks just as good as every other path, since the characteristics of those paths being evaluated are the same. Thus, keeping the selection criteria constant, any particular scheduler begins to service the jobs (more or less) in a round robin fashion.

Fortunately, this situation is much less likely to occur in a multiprogramming system than in a dedicated environment. Single programs often have *identical* control paths, generated by the outer nested parallel loops (the horizontal brackets in Chapter 3). With multiprogramming, a much wider variety of relative critical path distances, processor requests, execution times, etc. is likely to be found, giving the run–time scheduler a basis for decision making.

Furthermore, not only does a multiprogrammed system inherently offer a richer environment of different jobs than a monoprogrammed one does, but also the probability of a dynamic system, or nonzero arrival rates, is enhanced. This leads to potentially even greater differences in such parameters as the dynamic critical path. This is true even if a common job is run often, since an earlier arrival will already have completed a portion of its DAG before the next job arrives, so that its starting nodes will carry smaller critical path tags than the current arrival. (Indeed, it is probably much more likely to have a dynamic environment *between*, not *within*, separate DAGs).

Table 4.16 is illustrative of such results in a dynamic environment. Unlike the previous example, where all of the jobs arrived simultaneously, this particular example is more complex, demonstrating the dynamic modification of the general

Table 4.16. Programs COUNT and FIGI, 8 Processor Machine, 25% Folding.

| | Global Statistics | | | | | | | | COUNT | | FIGI | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SCH | TEX | EXE | STQ | BLK | EXT | CMP | TTT | PRC | PTT | ETS | PTT | ETS |
| 1 | 28189 | 5 | 1 | 262 | 166 | 215 | 9804 | 5 | 19812 | 3847 | 1073 | 1001 |
| 2 | 19592 | 7 | 5 | 204 | 166 | 292 | 5470 | 7 | 10752 | 3311 | 1685 | 956 |
| 3 | 21274 | 5 | 25 | 233 | 148 | 839 | 7285 | 6 | 13484 | 4084 | 5196 | 4700 |
| 4 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 5 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 6 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 7 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 8 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 9 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 10 | 22448 | 6 | 3 | 226 | 165 | 266 | 6847 | 6 | 13732 | 3727 | 1489 | 1016 |
| 11 | 25371 | 5 | 2 | 231 | 165 | 245 | 7851 | 5 | 15665 | 3697 | 1300 | 1294 |
| 12 | 28709 | 4 | 19 | 270 | 165 | 885 | 10948 | 5 | 22129 | 5742 | 5438 | 5406 |
| 13 | 25311 | 5 | 3 | 225 | 167 | 256 | 7657 | 5 | 15241 | 3309 | 1361 | 1279 |
| 14 | 28709 | 4 | 19 | 270 | 165 | 885 | 10948 | 5 | 22129 | 5742 | 5438 | 5406 |
| 15 | 25311 | 5 | 3 | 225 | 167 | 256 | 7657 | 5 | 15241 | 3309 | 1361 | 1279 |
| 16 | 28796 | 4 | 1 | 270 | 166 | 188 | 10292 | 5 | 22216 | 5759 | 967 | 933 |
| 17 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 18 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 19 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 20 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 21 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 22 | 19556 | 6 | 44 | 201 | 151 | 1257 | 6373 | 7 | 10849 | 3310 | 7878 | 7814 |
| 23 | 28796 | 4 | 1 | 270 | 166 | 188 | 10292 | 5 | 22216 | 5759 | 967 | 933 |
| 24 | 19592 | 7 | 5 | 204 | 166 | 292 | 5470 | 7 | 10752 | 3311 | 1685 | 956 |

collection of DAGs the run–time scheduler is examining.

For this example, an 8 processor machine with 25% folding was selected. Five copies of the program COUNT were scheduled to be run sequentially, arriving at the start of the simulation. One hundred and twenty copies of the program FIGI were dynamically added in groups of four, with arrival times spread out across an interval of 14976, in an attempt to keep a relatively constant flow of jobs into the system until near the end.

And once again, the pattern is still present. The largest dynamic critical path and volume algorithms (nos. 4–9, 17, and 18–22) return consistent and significant results on dynamic DAGs, with a speedup for this particular example of 30.6%. The smallest dynamic critical path and volume algorithms (nos. 16 and 23) again perform poorly, with an increase over the random scheduler's total execution time of 2.2%. All other schedulers are scattered somewhere in between these two values.

So, to summarize some of the concepts to this point, a run–time scheduler will make a difference in the throughput of a DAG or DAGs under the following conditions. First, the processors must be stressed. Placing a heavy load upon the processors forces the scheduler to make a decision as to which group of tasks will not currently be able to run. If that decision is wise, then good performance will result. Otherwise, the scheduler could perform worse even than random. Light loading of the processors reduces that pressure, with the limiting case being that all schedulers act the same, and approach an optimal throughput.

Second, a relatively large number of starting nodes must be presented to the run–time scheduler. Multiple independent control paths contribute to this factor.

The higher the number of nodes, the lower the probability that any two schedulers will select the same node, and thus yield the same results.

Third, these nodes should present different values to the scheduler (for whichever characteristic it is evaluating). Otherwise, the scheduler has no criteria upon which to form the basis of its decisions. For example, if all of the starting tasks have the same dynamic critical path tag value, then a dynamic critical path scheduler will select the highest one, until they are all equal, and then begin a round robin dispatch between all of the various paths.

Note that all three of these points are more likely to occur in a multiprogrammed operating system than on one which implements only monoprogramming. There are more demands on the processors, there are less dependence arcs between programs, and the dynamic arrivals, as well as the different types of jobs, contribute to a wider variety of nodes in the system.

And under these situations, it appears that the dynamic critical path and volume algorithms perform very well. Consistent and significant speedups are recorded over a random selection of tasks from the starting queue.

Graphically, the situation can be illustrated in Figure 4.1. Figure 4.1 shows a typical job mix in a multiprogramming system, i.e., one (or a few) very long jobs, with lots of smaller jobs also competing for resources.

In such situations, competition for resources often cause nodes to get "bumped off" consideration for the next node to be started. But which one (or ones) should lose out in Figure 4.1? Clearly, it seems prudent to give precedence to the nodes on

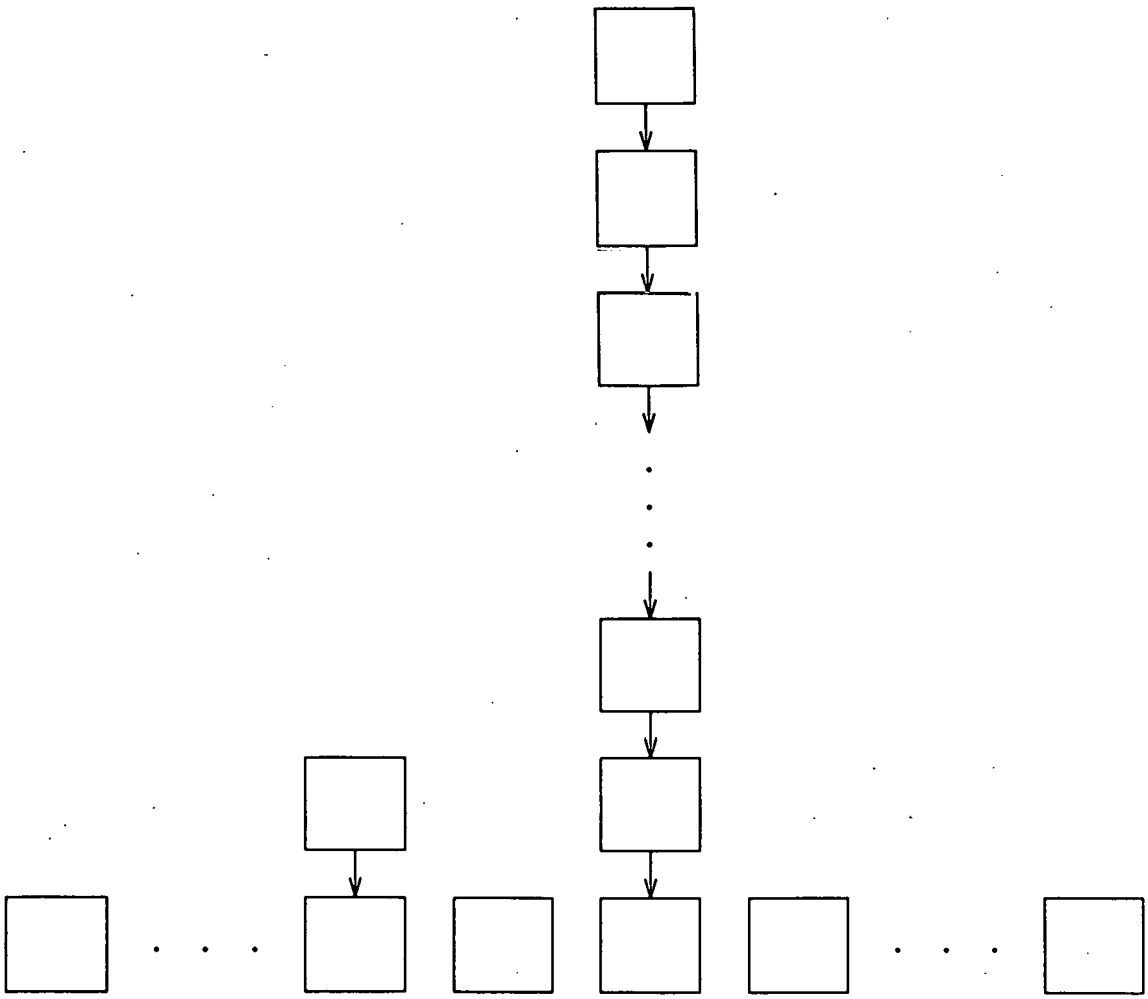


Figure 4.1. Multiprogramming DAG Collection.

top of the largest dynamic critical path and try to overlap the shorter nodes in parallel with the execution of the longest job. Any other choice is almost certain to be detrimental to system throughput.

Furthermore, the higher the relative difference in critical path heights, in conjunction with a larger number of small jobs requesting service from the processors, the more important a scheduler's decisions become to throughput. After all, with a low probability of "finding" an extremely high path in a huge sea of short nodes with schedulers such as random, greedy or generous execution times, etc., a greater potential exists for poor results. Therefore, in such situations, schedulers such as the dynamic critical path or volume algorithms are worth the investment in overhead more sophisticated schedulers require on real data programs.

One of the interesting results from the examples run was that the dynamic critical volume algorithms (nos. 18–22) did not outperform the dynamic critical path algorithms (nos. 4–9 and 17). Indeed, differences of only a few percent were usually recorded. This is especially surprising when folding was permitted since, as was pointed out in Section 3.2.4, use of the 25% folding rule means that the dynamic critical path distance for any arbitrary node may be off by a factor of four from the real time needed to execute that path. It was originally thought that since the sum of the space–time products more accurately represents the amount of work needed to complete some path than does the sum of the execution times, then the former would provide a superior selection criteria for the run–time scheduler. Such does not appear to be the case.

The reason this seems to be true is that there is no correlation between which path a node happens to reside upon, and the probability of that node's successors being folded. Stated another way, averaged over all nodes, paths with higher

dynamic critical tag values (tend to) have more nodes within those paths. The larger the number of nodes in a path, the higher the probability that some of those nodes will be forced to fold processors into time, thus increasing the "effective" dynamic critical path to an even larger value than it was before. Thus, in a sense, the dynamic critical path value does take into account the problem of folding, even if it is only a "side effect" of the probabilistic distribution of execution times and processor requests among the nodes.

One final comment on this subject should be made. Since the dynamic critical volume algorithms require a multiplication which is not needed in the dynamic critical path algorithms, the former series will obviously run slower than the latter. And since the dynamic critical path algorithms seem to work just as well, those algorithms are probably the ones which should be selected, given a choice between the two sets.

Another point of interest was that the secondary evaluation criteria in the event of ties in the critical path and volume algorithms did not have a consistent and significant effect. Usually, little difference was recorded, and it was not uncommon for the results to match exactly between different variations of the tie breakers. In those few instances where small differences did occur, no consistent pattern developed as to which criteria was best.

Also, examples were run on larger processor machines. When the number of processors was increased without changing the job mix, the obvious results were observed. That is, there were decreases in the total execution times and a lowering of

the relative differences between the schedulers, in a fashion similar to the transition from Table 4.13 to Table 4.14, and for the same reasons previously cited. On the other hand, scaling the number of processors and the degree of multiprogramming up by the same factor yielded virtually the same results as on the smaller machine. For instance the example shown in Table 4.13 was scaled up to a 32 processor machine, with four copies of the large program, and 280 copies of the small program. The same trends held. Random returned a total execution time of 61395. Smallest dynamic critical path and volume returned 69805 and 69797, respectively, which were the worst of all of the schedulers. The largest dynamic critical path and volume algorithms averaged a total execution time of 36270 and 36370, respectively.

And finally, the amount of folding permitted returned the same results as in Section 4.1, i.e., folding has a dramatic effect on the overall execution times, but shows no correlation between selection of a run–time scheduler. Readers are therefore referred to the work by Xu and Yew mentioned in previous sections for any further discussion.

### 4.2.2. Turnaround Time

To this point, it has been shown that the dynamic critical path and volume series of algorithms perform very well in a multiprogrammed environment, i.e., a substantial increase in speedup is recorded over random and most other schedulers. As can be noted from the far right columns in all of the examples cited, however, this speedup is gained at the expense of extreme turnaround times for the shorter programs.

This is a bad situation. For it is probably true that the shorter the program, the more users are interested in quick response time. The dynamic critical path algorithms certainly fail this test.

Why is this the case? The problem arises due to the inherent nature of the dynamic critical path algorithms in that they actually *avoid* terminal nodes. That is, if a program DAG is almost completed, and the run–time scheduler suddenly discovers a new starting node with a higher dynamic critical path value (due to either a new arrival or an existing DAG disposing of dependence links as predecessor nodes terminate) then the DAG with the largest critical path value is given preference over the shorter ones. This is by intent, of course, in order to maximize throughput.

The problem with this scheme, however, is that the closer a job gets to finishing, the lower the probability becomes of that same job being given access to the processors by the scheduler. It is clear that a paradox has been created, namely, jobs are only given the chance to complete if they are not close to completing. The implications of this strategy upon program turnaround time are obvious.

The solution to this problem seems to be one of balance. Obviously, techniques which go to extremes in either direction are not acceptable in a real system, and the critical path schedulers discussed in the previous section must be modified somehow in a manner which will take both parameters into account. That is, give priority to large critical path values most of the time in order to aid throughput, but once a job becomes "close" to finishing, then try and quickly flush it out of the system in order to aid turnaround time.

This section attempts to analyze that problem. The final two run-time schedulers listed in Table 3.2 are examined and compared in the standard fashion to random, etc. However, since many of the "side issues" discussed in Section 4.2.1 (e.g., the degree of multiprogramming, the effects of folding, etc.) yield the same results as they did in the previous section, those arguments are not repeated here.

The first algorithm to be discussed in this section is the one in row 24 of Table 3.2, which for lack of a better name will be known simply as "number 24". Scheduler number 24 first determines which node has the largest overall dynamic critical path tag value. Next, the earliest program number of that same type of program is located. (Program types are defined in Section 3.2.5, 3.2.6, and the Appendix). Finally, the largest dynamic critical path within that particular program is selected for execution.

Basically, what number 24 is trying to do is to combine the dynamic critical path scheme with FIFO, in an attempt to make some tradeoffs between throughput and turnaround time. It moves jobs which have been in the system for a long time towards completion. Furthermore, it avoids the common problem the dynamic critical path series have of processing all of the high paths first, until everything is of even height, and then providing approximately equal service to all possible paths in a round robin fashion, i.e., as soon as any path gets lower than any of the other paths it is ignored. On the other hand, because two of the three stages in number 24 are based upon the dynamic critical path, it still attempts to service the DAG with minimal loss of throughput.

And, in fact, number 24 actually works quite well. Consider for example the program set discussed in Table 4.13. (Although all of the copies of COUNT arrived simultaneously in Table 4.13, number 24 arbitrarily decides that which program is the "first" is the one with the lowest program number. And in real systems, all programs generally are assigned a unique program ID, which would be sufficient for number 24 to operate in the event of arrival time ties in the real world.)

For that particular collection of DAGs, number 24 returned a total execution time of 41640, or approximately a 30.9% speedup over random. With the exception of the critical path and volume algorithms, this result is better than any of the other schedulers. On the other hand, the critical path schedulers returned speedups in the range of 40%, which obviously is better than number 24.

But what price did the dynamic critical path schedulers pay for their extra speedup? Consider scheduler number 5 (largest dynamic critical path with ties broken by the smallest processor request), which returned the best throughput in this particular example. Scheduler number 5 showed an average execution time span for program COUNT of 32420. This is 26.7% worse even than random in this example. Similarly, the average turnaround time for COUNT with scheduler number 5 was 30.1% worse than random. (On the other hand, the execution time span and the turnaround time for program THREEDH were about 42% faster than random).

Compare that with the results obtained by scheduler number 24. Scheduler number 24 had an average execution time span for COUNT of 3318, and an average turnaround time of 23556. This is 87% and 11.6% faster, respectively, than random,

and 89.8% and 32.3% faster, respectively, than scheduler number 5. (The statistics for THREEDH were roughly the same as that returned by scheduler number 5). So at the cost of losing 10.9% (out of 41.8%) of the speedup afforded by scheduler number 5, scheduler number 24 returns an execution time span for the smaller program which is an order of magnitude smaller than number 5, and a turnaround time which is faster by a third.

Similar results were obtained for all other simulations conducted. For example the dynamic critical path algorithms in Table 4.16 (with nonzero arrival times) took 19556 time units to execute, while scheduler number 24 required 19592. These are speedups over random of 30.6% and 30.5%, respectively, which are obviously very close. But the dynamic critical path algorithms bought that extra 0.1% at a very heavy price. The average execution time span for FIGI was 933, and the average turnaround time was 967. (These values, execution time span and turnaround time, are much closer than before, since all copies of FIGI did not arrive simultaneously at the start of simulation, unlike the example cited in Table 4.13). This represents an 88% and an 87.7% speedup, respectively, over the dynamic critical path algorithm. Certainly this was a worthwhile tradeoff!

And so it seems that scheduler number 24, all things considered, performs quite well. Unlike the dynamic critical path algorithm, number 24 flushes out jobs that are about to complete, and avoids the pitfall of ignoring short DAG segments. Throughput does sometimes suffer (as was especially true in the first example cited) but the benefits of quick response time more than compensate for that loss.

The problem with scheduler number 24, of course, is that it requires classification of the programs submitted to the operating system by job type. This is a serious drawback. Furthermore, all of the program nodes are required to carry along that program type tag (or at least a pointer to a tag) which is not very space efficient. The next scheduler is an attempt to avoid those drawbacks.

The dynamic critical ratio algorithm is an attempt to balance throughput and turnaround time based only upon the characteristics of the DAG alone. No requirements are needed for job identification or typing.

The dynamic critical ratio algorithm simply takes the largest critical path value, unless some node is shorter than that value (or the value of the *last* node dispatched), in which case the shortest critical path value is used. This algorithm thus attacks the same problem as number 24, but by a different method. That is, large dynamic critical path values are normally used in an attempt to maximize throughput, but once a DAG segment gets "close" to completing, then it is given priority and pushed out of the system. "Close" in this instance is defined by the ratio value selected, which will be referred to simply as "R". (See Sections 3.2.5 and 3.2.6 for further information).

What type of results does the dynamic critical ratio algorithm return? Well, the answer to that question depends upon the value of R selected. For example, Figure 4.2 demonstrates how the total execution time, the program turnaround time, and the execution time span for the shorter program (FIGI, in this case) are affected by the value of R used for the example first cited in Table 4.16. (The specific values

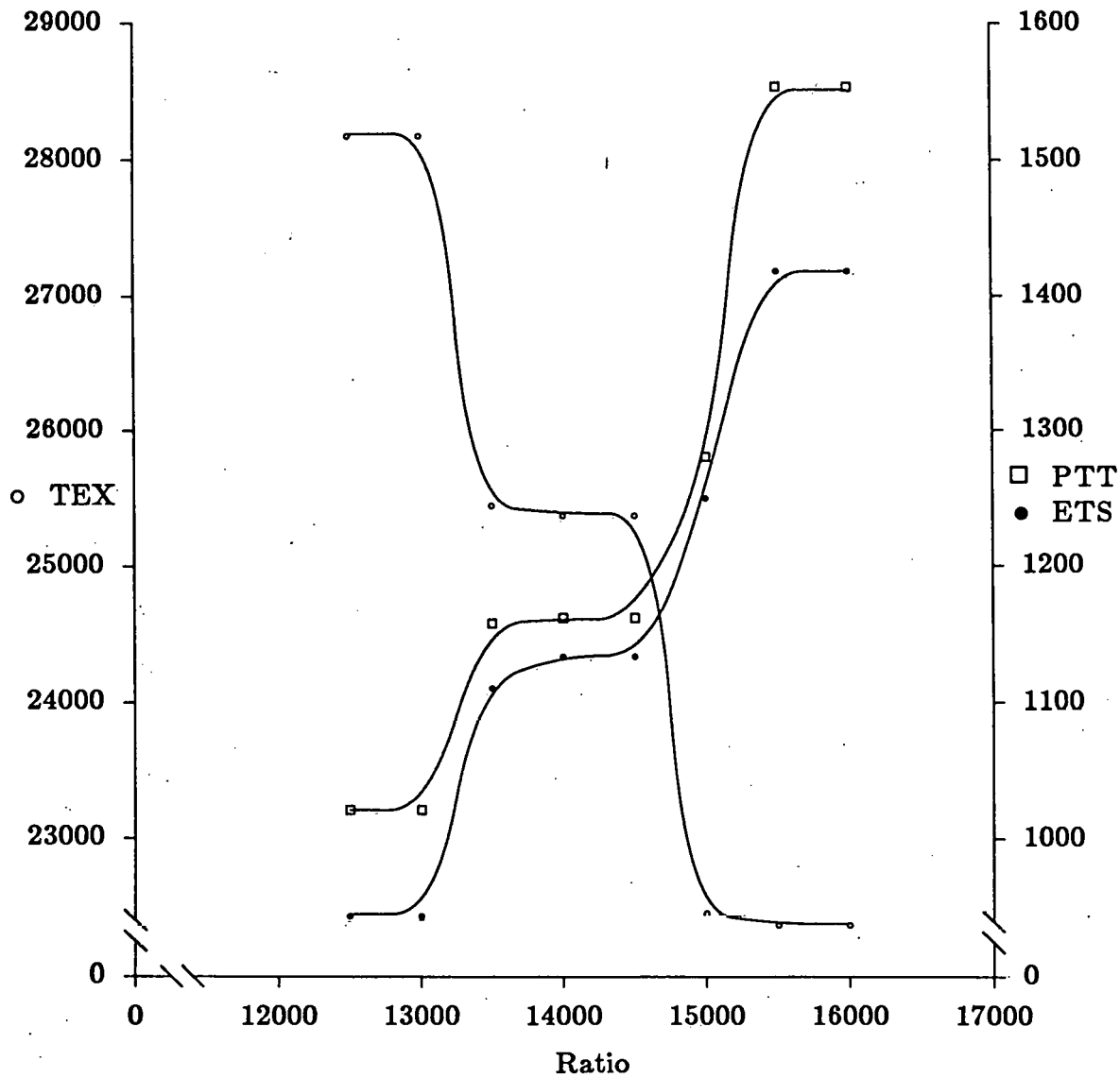



Figure 4.2. Performance Parameters as a Function of R.

used to generate Figure 4.2 for different values of R are reproduced in Table 4.17).

There are two main points of interest in Figure 4.2. First, it is clear that the total execution time is inversely proportional to the turnaround time and the

execution time span. (Since the turnaround time and execution time span curves follow each other, only one, the turnaround time curve, will usually be referred to in the rest of this section).

Obviously, this is to be expected. For as the dynamic critical ratio algorithm increasingly favors flushing smaller jobs out of the system, it increasingly helps the average turnaround time, but at the cost of increased overall system time.

The best results would be obtained at some point in the middle of the crossover between the curves. (In this example, the turnaround time curve started to drop significantly faster than the execution time curve started to rise on the right portion of the curves). For example, if an R value of 15000 is used in this particular example, total execution times of about 20.6% faster than random are recorded. While that is certainly less than what was recorded by the dynamic critical path algorithm

Table 4.17. Programs COUNT and FIGI with Variable Ratios, 8 Processor Machine, 25% Folding.

| | Global Statistics | | | | | | | | THREEDH | | COUNT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| R | TEX | EXE | STQ | BLK | EXT | CMP | TTT | PRC | PTT | ETS | PTT | ETS |
| 12500 | 28190 | 5 | 1 | 264 | 166 | 206 | 9854 | 5 | 20030 | 4063 | 1021 | 945 |
| 13000 | 28190 | 5 | 1 | 264 | 166 | 206 | 9854 | 5 | 20030 | 4063 | 1021 | 945 |
| 13500 | 25457 | 5 | 2 | 247 | 166 | 221 | 8383 | 5 | 17113 | 3983 | 1158 | 1112 |
| 14000 | 25387 | 5 | 2 | 246 | 164 | 223 | 8325 | 5 | 16801 | 3718 | 1162 | 1135 |
| 14500 | 25387 | 5 | 2 | 246 | 164 | 223 | 8325 | 5 | 16801 | 3718 | 1162 | 1135 |
| 15000 | 22473 | 6 | 2 | 238 | 166 | 234 | 7192 | 6 | 14555 | 3642 | 1280 | 1251 |
| 15500 | 22384 | 6 | 4 | 206 | 167 | 279 | 6256 | 6 | 12480 | 3308 | 1553 | 1420 |
| 16000 | 22384 | 6 | 4 | 206 | 167 | 279 | 6256 | 6 | 12480 | 3308 | 1553 | 1420 |

(or even number 24), the average program turnaround time and execution time spans for FIGI were 83.8% and 84% faster than the dynamic critical path algorithm, respectively. Furthermore, by decreasing the value of R even further into the areas of nonzero slopes, more complete tradeoffs between throughput and turnaround time can be made, depending upon the particular needs of the users.

The second point of interest in Figure 4.2 is that all of the curves show three distinct regions of stability. Why did this occur? Figure 4.3 helps to explain this phenomenon.

Assume that the column heights in Figure 4.3 represent the dynamic critical path values of some programs. Figure 4.3 then shows a very long program, a short program which has been in the system for some time (the dashed box represents the portion of the DAG which has already been completed) and the arrival of another short program DAG.

The value of R can then fall into three distinct regions, as shown in the figure. If R1 is used, then the dynamic critical ratio algorithm acts exactly like the largest dynamic critical path algorithm. Large critical path values are selected, short critical path values are ignored (except when the big programs can't run due to dependence links from currently executing predecessors), throughput is maximized, as is turnaround time. This is represented by the right portion of the graph in Figure 4.2.

If R3 is used, then the dynamic critical ratio algorithm acts exactly like the smallest dynamic critical path algorithm. Small critical path values are selected over large ones, and throughput and turnaround time are minimized. This is represented

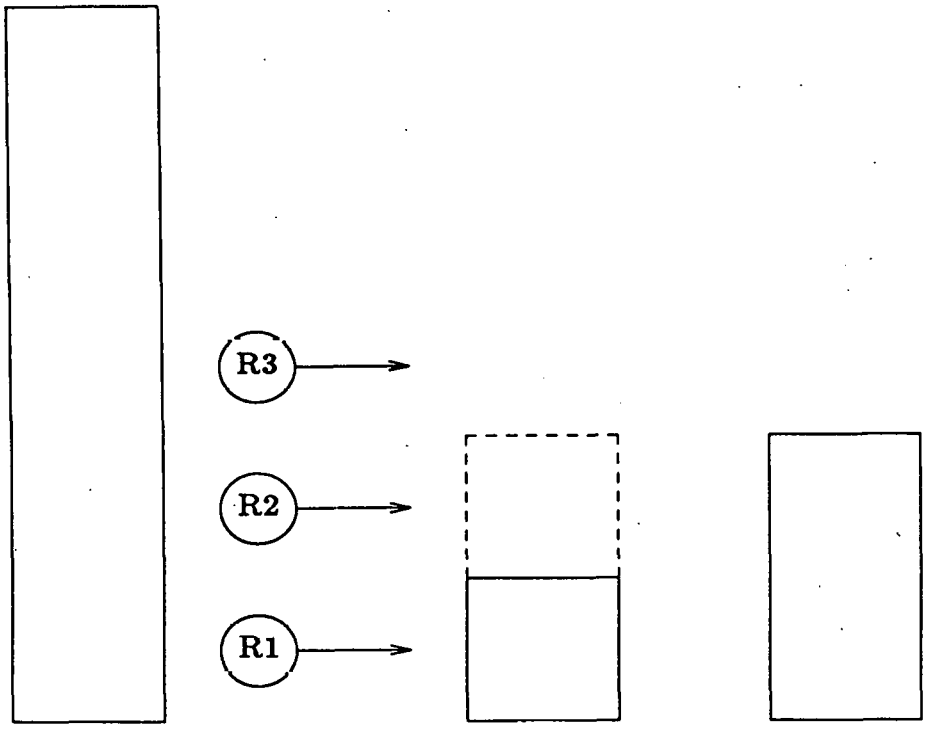



Figure 4.3. Possible Ranges for R.

by the left portion of the graph in Figure 4.2.

If R2 is used, then things are much different. Small DAGs close to completing are given priority, which helps turnaround time. When no small jobs are present (or they are blocked by dependencies from executing predecessors) then the largest dynamic critical path values are used, which helps throughput. New arrivals of short jobs are given the lowest priority, and must wait to run until dependencies block everyone else. The exact statistics returned when R is in the R2 region depends upon

the specific value of R, the average height of the program DAGs, and the average arrival rate of the DAGs.

As an example of this, consider Table 4.13 again. There, unlike the example just discussed, all of the programs arrived simultaneously. In such cases, R2 regions do not exist. The net effect of that situation is that the dynamic critical ratio algorithm essentially "breaks", and the analogous graph of Figure 4.2 would look like a step function if it was to be plotted. A binary choice is thus the only possible consideration without R2, i.e., the scheduler will act like either the largest or the smallest dynamic critical path algorithm. (Note that this situation can be simulated by very rapid arrival rates, i.e., if programs arrive faster than the scheduler can process them down below R. If that happens, then the algorithm becomes overloaded, and enters the R1 region. All of the problems previously discussed concerning the largest dynamic critical path algorithm then apply here, i.e., good throughput but bad turnaround time. Table 4.13 is the limiting case of this scenario, of course).

Assuming dynamic DAGs, however, the question then becomes: how should the value of R be selected? Unfortunately, there is no known way of automatically selecting this value. Generally, R should be selected such that it is less than the average height of the average small program entering the system (to avoid moving into region R3) but not so small so that the processors cannot "shrink" the height down below R before the next batch of jobs arrive (to avoid moving into region R1). Even though it tries to adapt by means of its two node working set, its sensitivity is still proportional to the size of that average R2 "window". Obviously, this is a func-

tion of the average arrival rate, the average service rate, the average height of jobs, etc., something that probably cannot be known *a priori*, without studying the job mix at a particular installation over a period of time.

Without that knowledge, however, the dynamic critical ratio algorithm is rather sensitive, and not robust. About the most that can be suggested at this point would be to have the value of R "tuned" on site depending upon the needs and demands of that system. Thus, as is true of all heuristics, it appears as if the *perfect* run–time scheduler which solves all possible problems under all possible circumstances does not exist.

# CHAPTER 5

# CONCLUSIONS

The last several decades have seen an enormous increase in computational capabilities. Originally, this was due to hardware innovations. More recently, the increase in speed has come about as a result of the use of increased parallelism.

In order to exploit this parallel hardware in general, and multiprocessors in particular, appropriate software must be developed which will take advantage of the underlying parallelism. For example, either parallel languages must be utilized, or compilers must be run which automatically detect parallel constructs in the code which can be executed concurrently on separate processors. And once those constructs have been recognized, a run–time scheduler is needed to determine which tasks should run in which order, and on how many processors.

This run–time scheduler faces many obstacles. For example, data dependencies specify a partial ordering upon the tasks, nodes may be bidimensional with both time and processor requests, those nodes may fold processors into time at processor allocation time, graphs are being modified at various times in a multiprogrammed, multitasked system, etc.

The general case is NP–hard. Nevertheless, the problem must be solved in the real world. Thus, an analysis of various heuristics is required.

Furthermore, real data taken from a working compiler and benchmark programs need to be studied. After all, analytical models are of limited practical significance, as their relationship to the characteristics of real programs, which are ultimately the things which must be executed, are not known *a priori*. This thesis attempts to examine a small portion of that problem.

Unfortunately, real programs contain thousands of nodes and dependence arcs. Especially in a multiprogrammed system, an optimal solution is therefore not feasible. A random scheduler, however, has been used as a practical (but not theoretical) worst case bound. Random is fast, and schedulers which do not perform substantially better than it are probably not a good choice.

Monoprogramming systems can utilize the services of a run–time scheduler to a lesser degree than can a multiprogramming system. There are several reasons for this. First, data dependencies are more of a problem in monoprogramming, while multiprogramming has more independent control paths to help create a larger number of starting nodes. The greater the number of starting nodes, the lower the probability that two different schedulers will select the same node, and thus yield the same results.

Second, single user systems stress the processors less. The more often schedulers can completely satisfy the requests of the starting queue, the less difference two different schedulers are likely to make. As a limiting case, in an unlimited processor environment, all schedulers act optimally.

Third, monoprogramming systems have less variety in the various paths. By definition, since individual programs comprise a subset of a multiprogramming system, the latter is bound to have a richer collection of parameters associated with its nodes. Such variety offers a real choice to schedulers, again making it less likely that different schedulers will select the same nodes, and therefore more likely that they will have a larger effect on the system performance parameters.

This is not to say that a run–time scheduler is irrelevant in a single user system. Indeed, some differences were recorded. What should be clear, though, is that it is more critical in a multiprogrammed environment, for the reasons cited above. Furthermore, it is more likely that many of the responsibilities of the scheduler can be moved to the compiler in a monoprogrammed system (such as calculation of the critical path tags), which is more static. This would allow a simpler, and thus faster, run–time scheduler to be implemented when only a single program at a time must be run.

In a multiprogramming system, the dynamic critical path algorithm seems to be the best choice for maximizing throughput. The dynamic critical path algorithm may be implemented by means of a separate scheduler, which calculates the largest sum of the execution times of a node's successors, and a dispatcher, which selects the node with the highest dynamic critical path value from the starting queue and allocates that node (with folding) on the processors. (The scheduler may be done at compile time *within* individual programs if it is known in advance that the entire program will arrive simultaneously).

This algorithm is a wise choice, because when some of the nodes cannot allocate processors due to competition for resources, it is detrimental to system throughput to avoid the largest dynamic critical path. Indeed, the smallest dynamic critical path algorithm usually performed even worse than random.

The dynamic critical volume algorithm, which is similar to the above except its tags are the largest sum of the product of the execution times and the processor requests of a node's successors, did not perform better than did the dynamic critical path algorithm. This was not expected when folding was permitted. However, it seems as if nodes with higher critical path values have a higher probability of getting folded, and thus also have a higher critical volume value. Thus, it appears as if the faster dynamic critical path algorithm is sufficient. Other parameters which had little or no consistent and significant effect upon the *choice* of schedulers include secondary evaluation criteria, the presence or absence of folding, etc.

Turnaround time is also very important to the users, particularly those that own small jobs. Unfortunately, the dynamic critical path algorithm ignores small jobs, maximizing throughput at the total expense of turnaround time. Algorithm number 24, which runs the largest dynamic critical path of the earliest program of the same type of program having the largest overall dynamic critical path, does very well. It combines the critical path technique, in an attempt to keep good throughput, with FIFO, in an attempt to balance good turnaround time. Although it provides lower throughput than the dynamic critical path algorithm, it is not substantially lower, and it more than makes up for the loss of throughput with dramatic improvements

in turnaround time.

Unfortunately, scheduler number 24 requires that programs somehow be sorted as to type. This may be an unrealistic restriction to make. The dynamic critical ratio algorithm attempts to avoid this restriction, while at the same time compromising throughput with turnaround time.

The dynamic critical ratio algorithm behaves differently, depending upon the ratio value and such factors as the job arrival rate. On one extreme, it acts like the largest dynamic critical path algorithm with good throughput but bad turnaround time, and on the other extreme, like the smallest dynamic critical path algorithm with bad throughput but good turnaround time. A window exists in the middle where throughput and turnaround time may be traded off. Unfortunately, the dynamic critical ratio algorithm is sensitive to its environment, and no way is currently known to automatically select the ratio value.

Several extensions can be made to this work. First, nodes were assumed to be nonpreemptive. This is not realistic, and this area should be investigated.

Second, the overhead inherent in the execution of the run–time scheduler and processor allocation has been ignored. To some extent, this is justified, i.e., tasks were made as large as possible by virtue of the course grain parallelism in Parafrase, which leads to a relative reduction of the contribution of overhead. Furthermore, certain aspects of the run–time scheduler can be run in parallel with other activity on the multiprocessor. Nevertheless, this area should be studied further.

Third, all DAGs were created equal, i.e., users could not specify their own priorities. In any real system, some priority scheme is essential.

Fourth, even though this work is closer to the "real world" than many analytical analyses, the next step would be to test some variations of a subset of these schedulers in a real operating system. This would allow for better analysis of job mix, arrival rates, program characteristics and variety, etc. than was possible in this thesis.

And finally, parameters important to selection of a good value for R in the dynamic critical ratio algorithm should be determined. It is expected that this will again require analysis of a real operating system, functioning over an extended period of time, in order to determine the appropriate characteristics under heavy, average, and light loading conditions.

# APPENDIX

This Appendix describes in detail the input parameters and output results produced by the scheduling simulator. The simulator was written in pascal. Input about the program DAGs is read from the file "graph". Simulation results are deposited into the file "results".

The first line of file graph must contain the following information, in order:

- The scheduling algorithm number. The specific scheduling algorithms and what they do are described elsewhere in this thesis. If the number corresponding to an algorithm is selected, the DAG is read once and scheduled once using that scheduling technique alone. If a value of 0 is entered here, the DAG in file graph is read and scheduled separately for each of the scheduling algorithms available.
- The total number of processors available. This specifies the machine size. All processors are assumed to be identical.
- The folding percentage requirements. This number specifies what percentage of the task's requested number of processors must currently be available in order for the task to be started. A value of 100 would turn folding off complctely.
- Trace switch. If a 1 is cntered here, then tasks are "traced" through the system, i.e., information about each task is printed whenever it changes states, along with the system time in which that change occurs. A value of 0 turns off this feature. The trace option allows the user to understand why things happen the

way that they do when all else fails. It does, however, produce a significant amount of output and should be used with caution.

For each node in the DAG, the following information must be provided, in order:

- The node arrival time. This is the time that this node enters the system, becoming a potential candidate for scheduling, and contributing to other nodes' dynamic critical path values, etc. These values must be monotonically increasing through the file graph. The node arrival time value must be on a separate line immediately preceding the following five values, which must be on the same line.
- The node number. This is essentially the "name" of the node. Values start at 1 and must increase by 1 throughout the file. (This provision was included in order to provide a consistency check on the input).
- The execution time for this task. This value may be changed internally by the scheduler due to folding considerations.
- The number of processors requested by this node. If this value exceeds the machine size, *forced folding* will be performed when the task is entered into the system in order to make the node fit. This will occur whether or not the run-time scheduler is permitted to fold. No other decision is possible if the DAG is to be run on the machine.

- Program type. Each type of program (i.e., different programs with identical DAGs) must be assigned a unique number. This value is used in a multiprogramming environment.
- Program number. Different programs of the same type must be assigned a unique number (within that type) in a multiprogramming environment. So, a three level hierarchy exists of: program types, program numbers, and node numbers.
- Predecessor list. On a single line following the above data, the node numbers of the immediate predecessors of this node are listed. Doubly linked dependence arcs will be created between all predecessor and successor nodes. If this node has a delayed arrival time, it is possible that some of its predecessors may have already been scheduled, executed, and left the system. Obviously then, those particular dependence arcs will not be created. This line must terminate with a 0.

The following information is placed in the file results after the simulation is complete:

- The scheduling technique. Both the required number and a brief textual description of that algorithm are written.
- The total number of processors available on the machine.
- The folding percentage requirements.

- The value of the trace option (and any of the results that option may have produced).
- The total time the simulation required. This is the internal "system time" needed to execute all of the nodes in the DAG using the specified scheduling technique. This value may be used as a measure of the throughput of the scheduling algorithm by comparing it with other values produced by different scheduling algorithms on the same DAG.
- The total number of nodes in the DAG.
- The average number of nodes executing on the processors.
- The average number of nodes on the starting queue, i.e., unable to run due to lack of sufficient processors.
- The average number of nodes blocked from execution (and the starting queue) due to predecessors with dependence links still in the system.
- The average turnaround time for each node. This is calculated from when the nodes first enter the system (their arrival times) until they have completed execution, released their processors, and left.
- The average completion time for each node. This is a measure of the time spent from when each node enters the starting queue (i.e., it is a candidate for execution) until it leaves the system.
- The average execution time for each node. This measures the time the nodes actually tie up the processors executing. (Any folding will obviously affect this

number).

- The average number of processors which are busy executing tasks.
- The processor work load. This value is the average of the sum of the number of processors busy plus the number of processors requested by the nodes on the starting queue. It attempts to gauge how many processors could be utilized by the DAG if they were available.

For each program type the following information is reported:

- The program type number.
- The number of programs of that type.
- The average turnaround time for the programs of that type (measured from when each of them first arrive in the system until they leave).
- The average execution time span for each of the programs of that type. This is calculated from when the first node of the program begins execution until the last node of that program has finished.

## REFERENCES

[AbDa86] Santosh Abraham and Edward Davidson. "Task Assignment Using Network Flow Methods for Minimizing Communication in n–Processor Systems", CSRD Report No. 598, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, Sept. 1986.

[BlDW86] Jacek Blazewicz, Mieczyslaw Drabowski and Jan Weglarz. *Scheduling Multiprocessor Tasks to Minimize Schedule Length.* **IEEE Transactions on Computers**, pp. 389–393, May 1986.

[Bokh79] Shahid Bokhari. *Dual Processor Scheduling with Dynamic Reassignment.* **IEEE Transactions on Software Engineering**, pp. 341–349, July 1979.

[CeKl83] Ruknet Cezzar and David Klappholz. *Process Management Overhead in a Speedup–Oriented MIMD System.* **Proceedings of the 1983 International Conference on Parallel Processing**, pp. 395–403.

[ChAb82] Timothy Chou and Jacob Abraham. *Load Balancing in Distributed Systems.* **IEEE Transactions on Software Engineering**, pp. 401–412, July 1982.

[ChKo79] Yuan–Chieh Chow and Walter Kohler. *Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System.* **IEEE Transactions on Computers**, pp. 354–361, May 1979.

[CHLE80] Wesley Chu, Leslie Holloway, Miu–Tsung Lan and Kemal Efe. *Task Allocation in Distributed Data Processing.* **Computer**, pp. 57–69, Nov. 1980.

[ChTs81] Francis Chin and Long–Lieh Tsai. *On J–maximal and J–minimal Flow–Shop Schedules.* **Journal of the Association for Computing Machinery**, pp. 462–476, July 1981.

[Coff76] Edward Coffman ed. **Computer and Job–shop Scheduling Theory.** John Wiley and Sons, New York, 1976.

[CoGr72] Edward Coffman and R. Graham. *Optimal Scheduling for Two–Processor Systems.* **Acta Informatica**, Vol. 1, No. 3, pp. 200–213, 1972.

[DBMS79] J. Dongarra, J. Bunch, C. Moler and G. Stewart. **Linpack User's Guide.** Siam Press, Philadelphia, 1979.

[ElHu80] Ossama El–Dessouki and Wing Huen. *Distributed Enumeration on Network Computers.* **IEEE Transactions on Computers**, pp. 818–825, Sept. 1980.

[GaJo79] Michael Garey and David Johnson. **Computers and Intractability: A Guide to the Theory of NP–Completeness.** W. H. Freeman & Co., San Francisco, 1979.

[GKLS83] Daniel Gajski, David Kuck, Duncan Lawrie and Ahmed Sameh. *Cedar – A Large Scale Multiprocessor.* **Proceedings of the 1983 International Conference on Parallel Processing**, Aug. 1983.

[GLPV83] Daniel Gajski et al. "Second Preliminary Specification of Cedar", Cedar Document No. 8, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, Feb. 1983.

[GoSc82] Allan Gottlieb and J. Schwartz. *Networks and Algorithms for Very–Large–Scale Parallel Computation.* **Computer**, pp. 27–36, Jan. 1982.

[GyEd76] V. Gylys and J. Edwards. *Optimal Partitioning of Workload for Distributed Systems.* **Digest of Papers, IEEE 1976 COMPCON Fall**, pp. 353–357.

[HoIr83] Lawrence Ho and Keki Irani. *An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment.* **Proceedings of the 1983 International Conference on Parallel Processing**, pp. 338–340.

[Hu61] T. Hu. *Parallel Sequencing and Assembly Line Problems.* **Operations Research**, pp. 841–848, Nov.–Dec. 1961.

[Hu82] T. Hu. **Combinatorial Algorithms.** Addison–Wesley Publishing Co., Reading, Massachusetts, 1982.

[Husm86] Harlan Husmann. "Compiler Memory Management and Compound Function Definition for Multiprocessors", CSRD Report No. 575, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, Aug. 1986.

[KDLS86] David Kuck, Edward Davidson, Duncan Lawrie and Ahmed Sameh. *Parallel Supercomputing Today and the Cedar Approach.* **Science**, Vol. 231, pp. 967–974, Feb. 28, 1986.

[KKLW80] David Kuck, Robert Kuhn, Bruce Leasure and Michael Wolfe. *The Structure of an Advanced Vectorizer for Pipelined Processors.* **Fourth**

**International Computer Software and Applications Conference**, Oct. 1980.

[KKPL81] David Kuck et al. *Dependence Graphs and Compiler Optimizations.* **Proceedings of the 8th ACM Symposium on Principles of Programming Languages**, pp. 207–218, Jan. 1981.

[KLVY82] David Kuck, Kyungsook Lee, Alexander Veidenbaum and Pen–Chung Yew. "Notes on Machine Control Structures", Cedar Document No. 3, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, April 1982.

[Kras72] Paul Kraska. "Parallelism Exploitation and Scheduling", Report No. 72–518, University of Illinois at Urbana–Champaign, Department of Computer Science, June 1972.

[KrWe85] Clyde Kruskal and Alan Weiss. *Allocating Independent Subtasks on Parallel Processors.* **IEEE Transactions on Software Engineering**, pp. 1001–1016, Oct. 1985.

[Kuck78] David Kuck. **The Structure of Computers and Computations.** John Wiley and Sons, New York, 1978.

[MaLT82] Perng–Yi Ma, Edward Lee and Masahiro Tsuchiya. *A Task Allocation Model for Distributed Computing Systems.* **IEEE Transactions on Computers**, pp. 41–47, Jan. 1982.

[Mill84] Allan Ray Miller. "Control Unit Performance Issues in a Multiprogrammed, Multiprocessing Computer", Report No. 84–1178, University of Illinois at Urbana–Champaign, Department of Computer Science, July 1984.

[PeZa86] Ronald Perrott and Adib Zarea–Aliabadi. *Supercomputer Languages.* **ACM Computing Surveys**, pp. 5–22, March 1986.

[Poly86] Constantine Polychronopoulos. "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems", CSRD Report No. 595, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, Aug. 1986.

[RaCG72] C. Ramamoorthy, K. Chandy and Mario Gonzalez. *Optimal Scheduling Strategies in a Multiprocessor System.* **IEEE Transactions on Computers**, pp. 137–146, Feb. 1972.

[RaSH79] Gururaj Rao, Harold Stone and T. Hu. *Assignment of Tasks in a Distributed Processor System with Limited Memory.* **IEEE Transactions on Computers**, pp. 291–299, April 1979.

[Sahn83] Sartaj Sahni. "Scheduling Supercomputers", Report No. 83–3, University of Minnesota, Computer Science Department, Feb. 1983.

[SBDG76] B. Smith et al. **Matrix Eigensystem Routines – Eispack Guide.** Springer–Verlag, Heidelberg, West Germany, 1976.

[Schw61] Eugene Schwartz. *An Automatic Sequencing Procedure With Application to Parallel Programming.* **Journal of the Assocation for Computing Machinery**, pp. 513–537, Oct. 1961.

[Stan85] John Stankovic. *An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling.* **IEEE Transactions on Computers**, pp. 117–130, Feb. 1985.

[StBo78] Harold Stone and Shahid Bokhari. *Control of Distributed Processes.* **Computer**, pp. 97–106, July 1978.

[Ston77] Harold Stone. *Multiprocessor Scheduling with the Aid of Network Flow Algorithms.* **IEEE Transactions on Software Engineering**, pp. 85–93, Jan. 1977.

[Ston78] Harold Stone. *Critical Load Factors in Two–Processor Distributed Systems.* **IEEE Transactions on Software Engineering**, pp. 254–258, May 1978.

[Veid85] Alexander Veidenbaum. "Compiler Optimizations and Architecture Design Issues for Multiprocessors", CSRD Report No. 520, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, May 1985.

[Wolf82] Michael Wolfe. "Optimizing Supercompilers for Supercomputers", Report No. 82–1105, University of Illinois at Urbana–Champaign, Department of Computer Science, Oct. 1982.

[XuYe83] Qing–Xian Xu and Pen–Chung Yew. "Queuing Analysis for a Multiprocessor System with Multiprogramming", Cedar Document No. 15, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, March 1983, Rev. June 1984.

[XuYe84] Qing–Xian Xu and Pen–Chung Yew. "Simulations and Analysis for a

Multiprocessor System with Multiprogramming", Cedar Document No. 30, University of Illinois at Urbana–Champaign, Center for Supercomputing Research and Development, Feb. 1984.

# VITA

Allan Ray Miller was born [redacted], [redacted] He graduated *magna cum laude* from the University of Central Florida (UCF) with a B.S. degree in Computer Science in March 1979. He attended UCF for one extra quarter as a graduate student while waiting for the start of the next semester at the University of Illinois at Urbana–Champaign. While attending UCF, he worked at the Experimental Computer Simulation Laboratory at the Naval Training Equipment Center (subcontracted through UCF). Ray Miller then attended the University of Illinois, receiving his M.S. degree in Computer Science in August 1984. ("Control Unit Performance Issues in a Multiprogrammed, Multiprocessing Computer", Report No. 84–1178, University of Illinois, Department of Computer Science, July 1984). Straight A's were received in all Computer Science courses while working towards the B.S. and M.S. degrees. He completed this Ph.D. thesis in Computer Science in May 1987. While at the University of Illinois, he was employed as a research assistant by the Department of Computer Science from August 1979 through December 1984. From January 1985 through graduation he was employed as a research assistant by the Center for Supercomputing Research and Development. He is a member of the following honor societies: Sigma Xi, Omicron Delta Kappa, and Tau Beta Pi. He is also a member of the Association for Computing Machinery and ACM SIGCAPH (special interest group in computers and the physically handicapped).

| BIBLIOGRAPHIC DATA SHEET | 1. Report No. CSRD-656 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|
| 4. Title and Subtitle<br>NONPREEMPTIVE RUN-TIME SCHEDULING ISSUES ON A MULTITASKED, MULTIPROGRAMMED MULTIPROCESSOR WITH DEPENDENCIES, BIDIMENSIONAL TASKS, FOLDING, AND DYNAMIC GRAPHS | | | 5. Report Date<br>May 1987<br>6. |
| 7. Author(s)<br>Allan Ray Miller | | | 8. Performing Organization Rept. No. CSRD-656 |
| 9. Performing Organization Name and Address<br>University of Illinois at Urbana-Champaign<br>Center for Supercomputing Research and Development<br>Urbana, Illinois 61801-2932 | | | 10. Project/Task/Work Unit No.<br>11. Contract/Grant No. NSF DCR84-10110 & NSF DCR84-06916; US DOE DE-FG02-85ER 25001; IBM Donation, Control Data Corp. |
| 12. Sponsoring Organization Name and Address<br>National Science Foundation, Washington, D.C.<br>U.S. Department of Energy, Washington, D.C.<br>IBM Corporation, Armonk, N.Y.<br>Control Data Corporation, | | | 13. Type of Report & Period Covered<br>Doctoral Dissertation<br>14. |

15. Supplementary Notes

16. Abstracts

Increases in high speed hardware have mandated studies in software techniques to exploit the parallel capabilities. This thesis examines the effects a run-time scheduler has on a multiprocessor. The model consists of directed, acyclic graphs, generated from serial FORTRAN benchmark programs by the parallel compiler Parafrase. A multitasked, multiprogrammed environment is created. Dependencies are generated by the compiler. Tasks are bidimensional, i.e., they may specify both time and processor requests. Processor requests may be folded into execution time by the scheduler. The graphs may arrive at arbitrary time intervals. The general case is NP-hard, thus, a variety of heuristics are examined by a simulator. Multiprogramming demonstrates a greater need for a run-time scheduler than does monoprogramming for a variety of reasons, e.g., greater stress on the processors, a larger number of independent control paths, more variety in the task parameters, etc. The dynamic critical path series of algorithms perform well. Dynamic critical volume did not add much. Unfortunately, dynamic critical path maximizes turnaround time as well as throughput. Two schedulers are presented which balance throughput and turnaround time. The first requires classification of jobs by type; the second requires selection of a ratio value which is dependent upon system parameters.

17. Key Words and Document Analysis. 17a. Descriptors

algorithms
operating-systems
scheduling
performance-evaluation
software

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

| 18. Availability Statement | 19. Security Class (This Report) | 21. No. of Pages |
|---|---|---|
| Release unlimited | UNCLASSIFIED | 107 |
| | 20. Security Class (This Page) UNCLASSIFIED | 22. Price |