




DEPARTMENT OF THE AIR FORCE

**AIR UNIVERSITY**

# AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-14



# Improved Task Scheduling for Parallel Simulations

THESIS

Andrew E. McNear
Captain, USAF

AFIT/GCS/ENG/91D-14



AFIT/GCS/ENG/91D-14

# Improved Task Scheduling for Parallel Simulations

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

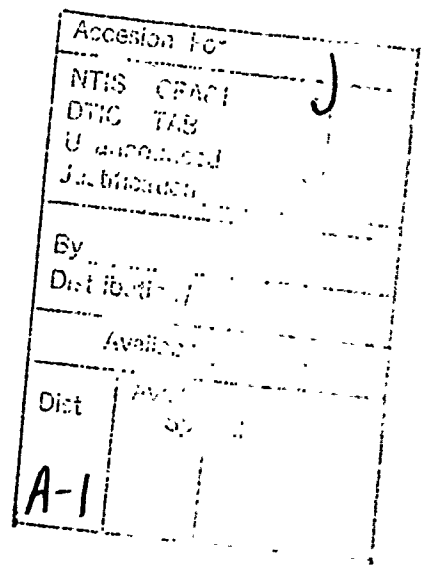
Requirements for the Degree of

Master of Science in Computer Engineering

Andrew E. McNear, B.S.E.E

Captain, USAF






December, 1991

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3 REPORT TYPE AND DATES COVERED |
|---|---|---|
| | December 1991 | Master's Thesis |

**4. TITLE AND SUBTITLE**
Improved Task Scheduling for Parallel Simulations

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Andrew E. McNear, Capt, USAF

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology, WPAFB OH 45433-6583

**8 PERFORMING ORGANIZATION REPORT NUMBER**
AFIT/GCS/ENG/91D-14

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Lt Col John Toole
DARPA
3701 N. Fairfax Dr
Arlington, VA 22203

**10 SPONSORING MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited

**12b DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The objective of this investigation is to design, analyze, and validate the generation of optimal schedules for simulation systems. Improved performance in simulation execution times can greatly improve the return rate of information provided by such simulations resulting in reduced development costs of future computer/electronic systems. Optimal schedule generation of precedence-constrained task systems including iterative feedback systems such as VHDL or war gaming simulations for execution on a parallel computer is known to be $NP$-hard. Efficiently parallelizing such problems takes full advantage of present computer technology to achieve a significant reduction in the search time required. Unfortunately, the extreme combinatoric 'explosion' of possible task assignments to processors creates an exponential search space prohibitive on any computer for search algorithms which maintain more than one branch of the search graph at any one time. This work develops various parallel modified backtracking (MBT) search algorithms for execution on an $iPSC/2$ hypercube that bound the space requirements and produce an optimally minimum schedule with linear speed-up. The parallel MBT search algorithm is validated using various feedback task simulation systems which are scheduled for execution on an $iPSC/2$ hypercube. The search time, size of the enumerated search space, and communications overhead required to ensure efficient machine utilization during the parallel search process are analyzed. The various applications indicate appreciable improvement in performance using this method.

**14. SUBJECT TERMS**
multiprocessor scheduling, parallel simulation, iterative task scheduling

**15. NUMBER OF PAGES**
XXX

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers, may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | | | |
|---|---|---|---|
| C | - Contract | PR | - Project |
| G | - Grant | TA | - Task |
| PE | - Program Element | WU | - Work Unit Accession No. |

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with..., Trans. of , To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

| | | |
|---|---|---|
| DOD | - | See DoDD 5230.24, "Distribution Statements on Technical Documents." |
| DOE | - | See authorities |
| NASA | - | See Handbook NHB 2200.2. |
| NTIS | - | Leave blank. |

Block 12b. Distribution Code.

| | | |
|---|---|---|
| DOD | - | Leave blank. |
| DOE | - | Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports. |
| NASA | - | Leave blank. |
| NTIS | - | Leave blank. |

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*

Blocks 17. - 19. Security Classifications Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.



★U.S.GPO 1990-0-273-271

## *Preface*

The purpose of this study is to investigate the complexities of generating optimally minimum schedules for war gaming and VHDL simulations using a parallel computer upon which the simulations are executed. These types of simulations generally exhibit arbitrary characteristics including feedback resulting in their classification as $NP$-hard problems. Additionally, simulation tasks can execute repeatedly. Therefore, exponential time requirements exist in attempting to find an optimal solution where the assignment of simulation tasks to processors results in minimal execution time. The general characteristics of simulation systems are analyzed to reduce the extreme combinatoric 'explosion' of the search process. Several methods are presented for minimizing the search space and time requirements using an $iPSC/2$ hypercube. The implementation is developed using the Ada programming language providing rapid prototyping of the design and ease of maintenance.

In analyzing the complexities of the scheduling problem, the two types of simulation applications considered, and the run-time environment of the $iPSC/2$ hypercube, several people provided immeasurable assistance. I'm deeply indebted to my thesis advisor, Dr Gary B. Lamont for his continued patience and guidance. His encouragement and advise throughout my research was instrumental in obtaining viable results. I also wish to thank my thesis committee members, Maj Christensen and Dr Hartrum for their insight into simulation applications and characteristics which guided me torwards meaningful and useful solutions to present simulation efforts. Also, without the help of Mr Rick Norris, the system administrator for the parallel processing cluster, validation of my parallel design would not have been possible. Finally, I wish to thank my loving family for their love and understanding during those long, sleepless nights and weekends when I was diligently working.

Andrew E. McNear

*Table of Contents*

## *List of Figures*

## *List of Tables*

*Abstract*

The objective of this investigation is to design, analyze, and validate the generation of optimal schedules for iterative parallel task systems with feedback constraints. The specific applications are large-scale VHDL circuit simulations and war gaming simulations which are designed to operate on message passing, parallel computers such as the Intel family of hypercubes. By creating optimal run-time schedules where the schedule length is optimally minimum, the maximum speed-up achievable given the characteristics of a simulation system can be achieved. Such improved performance in simulation execution times can greatly improve the return rate of information provided by such simulations resulting in r[illegible]ced development costs of future computer/electronic systems.

Optimal schedule generation of precedence-constrained task systems including iterative feedback systems such as VHDL or war gaming simulations for execution on a parallel computer is known to be $NP$-hard in all but the most trivial cases. Consequently, large search spaces must be explored at a considerable time expense even for the most powerful single processor computers. Efficiently parallelizing such problems takes full advantage of present computer technology to achieve a significant reduction in this time requirement. Unfortunately, generating optimal schedules requires implicit examination of all possible solutions. This coupled with the extreme combinatoric 'explosion' of possible task assignments to processors creates an exponential search space prohibitive on any computer for search algorithms which maintain more than one branch of the search graph at any one time. This work develops various parallel modified backtracking (MBT) search algorithms for execution on an $iPSC/2$ hypercube that bound the space requirements and produce an optimally minimum schedule with linear speed-up. Although the logical search space remains exponential, the physical memory space required for the search process remains within the physical memory constraints of the computer.

To validate the parallel MBT search algorithm developed, various feedback task simulation systems are scheduled for execution on an $iPSC/2$ hypercube. The search time, size of the enumerated search space, and communications overhead required to ensure efficient machine utilization during the parallel search process are analyzed. The various applications indicate appreciable improvement in performance using this method.

# Improved Task Scheduling for Parallel Simulations

## *I. Introduction*

### *1.1 Motivation for Parallel Processing*

Computer architecture has evolved greatly over the last decade with the proliferation of parallel computer research and development efforts. The driving force behind such investments being the potential speed-up of solving complex and computationally intensive problems such as wave mechanics, fluid dynamics, and structural analysis [16], Very Large Scale Integration (VLSI) circuit simulations [23] [9], and real-time assignment based problems within the Strategic Defense Initiative (SDI) research area [18]. The potential speed-up using a parallel, von-Neumann based computer can be defined as follows [11]:

$$S_p = \frac{T_1}{T_p} \tag{1.1}$$

where $T_1$ is the execution time for the best serial algorithm on a single processor, and $T_p$ is the execution time for a parallel algorithm using $P$ processors. Thus, if a sequential algorithm to simulate a VLSI circuit through 10 seconds of activity takes 120 minutes to complete when implemented on a single processor, applying the same simulation using an effective parallel algorithm on a parallel computer with 120 processors would theoretically take 1 minute! Unfortunately, it was suggested by Amdahl that for a program with serial work fraction $s$, the maximum parallel speed-up obtainable is bounded by $1/s$. Therefore, if a given program contains a 5% serial work fraction, then the largest achievable speed-up is 20 regardless of the number of processors used. If $P$ is the number of processors, $s$ is the amount of time spent by a serial processor on serial parts of a program, and $1 - s$ is the amount of time spent by a serial processor on parts of a program that can be done in parallel, then Amdahl's law gives the parallel processing time

$$T_p = sT_1 + \frac{(1-s)}{P}T_1 \tag{1.2}$$

resulting in the speed-up

$$S_p = \frac{P}{1 + (P-1)s} \tag{1.3}$$

Thus, the $\lim_{P\to\infty} S_p = 1/s$. The derivative of this equation with respect to $s$ reveals a very interesting property for small $s$.

$$\frac{dS_p}{ds} = \frac{-P^2+1}{(1+(P-1)s)^2} \tag{1.4}$$

$$\lim_{\frac{dS_p}{ds}\to 0} = -P^2+1 \tag{1.5}$$

As figure 1.1 shows, the degenerative speed-up curve has a slope of $-P^2$ as $s \to 0$, and $S_p$ quickly drops off for small increases in $s$ – a very undesirable characteristics when attempting to speed-up a program's performance on a parallel machine. Fortunately, parallel applications tend to scale with the available computing power; *i.e.*, given more computing power as with a parallel computer, the applications are expanded (more data items, for example) to utilize the available hardware resources. For instance, doubling the number of processors allows doubling the number of data variables to ensure comparable utilization of the physical system. Also, the time for program loading, serial bottlenecks, and I/O that make up the $s$ component of an application do not scale with problem size.

This idea of scaled speed-up leads to the definition of an effective parallel algorithm as one in which $\lim_{n\to\infty} s(n) = 0$ where $s(n)$ is the fraction of the sequential algorithm that must be run sequentially (not parallelizable) and is dependent upon the problem size $n$. Thus, the speed-up of an effective parallel algorithm when applied to large problems has a limit which is defined as linear speed-up [13:5]:

$$S_p = \frac{p}{1+(p-1)s(n)} \tag{1.6}$$

$$\lim_{n\to\infty} S_p = p \tag{1.7}$$

When considering the inverse of Amdahl's paradigm, the attractiveness of this reasoning becomes apparent. Rather than ask how fast a given serial program would run on a parallel processor, we ask how long a given parallel program would have taken to run on a serial processor. Using the variables as before, a uniprocessor requires time $s + (1-s)P$ to complete a program where the second term represents the serialized parallel time. This leads to the scaled speed-up equation

$$S_{scaled} = \frac{s+(1-s)P}{s+(1-s)} = P+(1-P)s \tag{1.8}$$

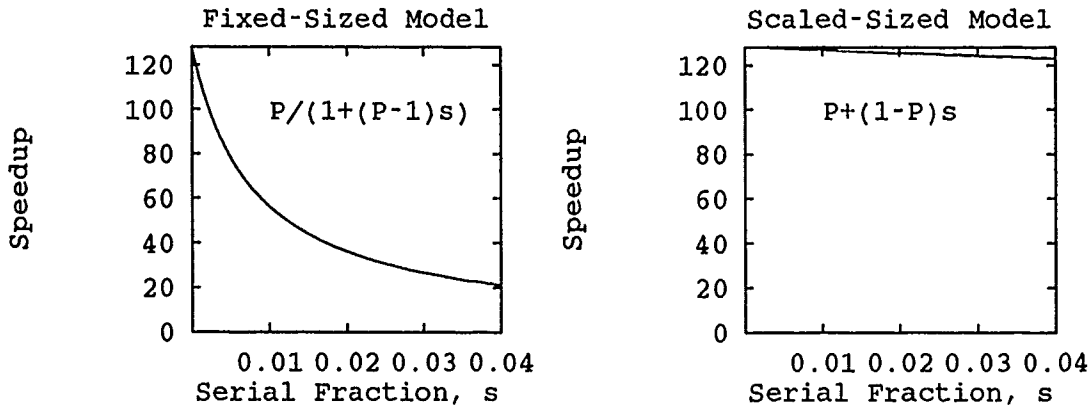



Figure 1.1. Comparative speed-up of Amdahl's law and problem scaling for P = 128.

As figure 1.1 shows, the advantages of such reasoning become apparent. The degenerative speed-up curve now has a linear slope of $1 - P$, a significant improvement in performance. Thus, when speed-up is measured by scaling the problem size, the serial fraction $s$ tends to shrink resulting in much better parallel performance than is implied by Amdahl's paradigm [16].

### *1.2 Parallel Applications*

The need for fast problem solvers is quite apparent. Weather prediction results are useless if the time to produce the prediction is greater than the time for the actual event to occur; *i.e.*, the event predicted to occur in the future has already occurred. The same is true in SDI research involving real-time assignment strategies of weapons to targets. The general assignment problem can be stated as the problem of assigning $N$ resources to $M$ consumers subject to some minimization or maximization constraint [23]. Therefore, the assignment problem within SDI involves the assignment of weapons to incoming targets such as ballistic missiles in such a way that the maximum number of targets with the potential for the greatest amount of damage are destroyed using the minimum amount of weapons. A solution to this problem is equally useless if the time taken to produce the solution is longer than the time for the incoming missiles to reach their points of destruction.

The development of VLSI circuits is very complex with many intermediate steps between design conception and production circuits. Prior to circuit simulation abilities, various stages of development required laboratory fabrication and testing – a very tedious, expensive, and time consuming process. With the advent of computer simulation languages such as the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language, or VHDL, circuit behaviors

can now be modeled on a computer, thus vastly decreasing the turnaround time from design to production. Unfortunately, VHDL simulations for current applications take a disproportionate amount of computer time when compared to the size of the circuit being modeled. When a parallel computer is used, the mapping strategy of assigning specific VHDL tasks to processors can have a significant impact on the simulation time.

Simulation efficiency is defined as the ratio of time of circuit behavior modeled, $T_{sim}$, and the time to model the circuit, $T_{run}$, for that amount of simulation time.

$$\epsilon_{sim} = \frac{T_{sim}}{T_{run}} \tag{1.9}$$

As Table 1.1 indicates, a circuit simulation of an 8-bit adder requiring 32 VHDL tasks on an Intel iPSC/2 hypercube computer with eight processors results in a simulation efficiency of 0.006% when using the 'level8' optimal static scheduling strategy, and 0.001% when using an unbalanced static scheduling strategy which approximates the results of a single processor simulation [38:5-4]. The computer architecture, simulation task dependencies, and the scheduling algorithm effectiveness both play crucial roles in determining speed-up of a particular parallel simulation.

Table 1.1. Run-time performance comparison of an 8-bit VHDL simulation

| 8-node mappings | Simulated Time, $T_{sim}$ | | | | |
|---|---|---|---|---|---|
| | *1000 nsec* | *8000 nsec* | *16000 nsec* | *32000 nsec* | *64000 nsec* |
| | Run Time, $T_{run}$ | | | | |
| level8 | 18 sec | 126 sec | 252 sec | 509 sec | 1108 sec |
| unbalanced8 | 79 sec | 573 sec | 1147 sec | 2272 sec | 4404 sec |

## *1.3 The Task Scheduling Problem*

Task scheduling in the context of this research is the problem of assigning $n$ precedence-constrained tasks to $m$ processors where $n \gg m$ such that the overall execution time is optimally minimal. The task systems considered are iterative feedback systems where feedback precedence-constraints can exist between multiple executions of the tasks. Unfortunately, static and dynamic iterative task scheduling lack any definitive optimization for parallel machine efficiency [38].

*1.3.1 Static versus Dynamic Iterative Task Systems* A static iterative task system is one in which assignment of the tasks to the processors of a parallel computer remain unchanged during

the execution of the tasks. In dynamic iterative task systems however, the structure of the task system changes during the execution process requiring an adjustment of the task schedule in order to maintain required performance. Figure 1.2 represents this difference. Initially, the static task system consists of two tasks in which each task executes twice. An optimal schedule results in a schedule which completes in three time units. This optimal schedule and the mapping of specific tasks to specific processors is determined prior to execution of the task system. In the dynamic system however, upon execution of the schedule, the first iteration of $T1$ generates the requirement for a new, previously unconsidered task, $T3$, to execute. Therefore, to ensure an optimal schedule is maintained, the scheduling algorithm must gain control of computer resources at time $t_1$ to generate the new optimal schedule shown. The task system is then directed to follow this new schedule to ensure optimal performance until such time that the system again changes.

*1.3.2 Parallel Simulation Implementation Considerations* When examined from an architectural point-of-view, several factors must be considered in attempting to achieve effective parallel performance for simulation systems:

- Load balancing of computations
- Communications overhead between processors
- Mapping strategy

Load balancing refers to the even distribution of the workload among the available processors. The communications overhead between processors refers to the associated time cost of having tasks which must communicate assigned to different processors. The communications time between tasks on a single processor is usually much smaller; however, due to system loading and operating system control, task intraprocessor communication could be greater. The mapping strategy refers to the way in which individual simulation tasks are assigned to processors. As previously shown in Table 1.1, such strategies can have a significant effect on the simulation efficiency.

In general, the mapping strategy determines run-time load balancing and communications overhead. For instance, if a system of equal execution time (EET) tasks is mapped onto a 1024 processor system where 80% of the tasks are assigned to 10% of the processors, an unbalanced condition of processor utilization occurs. If however, the tasks exhibit a strong precedence-constraint relationship, such load unbalancing may be unavoidable within the domain of that task system; *i.e.*, the maximum achievable scheduling parallelization produces poor machine utilization. Also, task systems with a strong precedence relationship requiring much communication among tasks may suffer significant communications overhead if many of the tasks which communicate don't reside

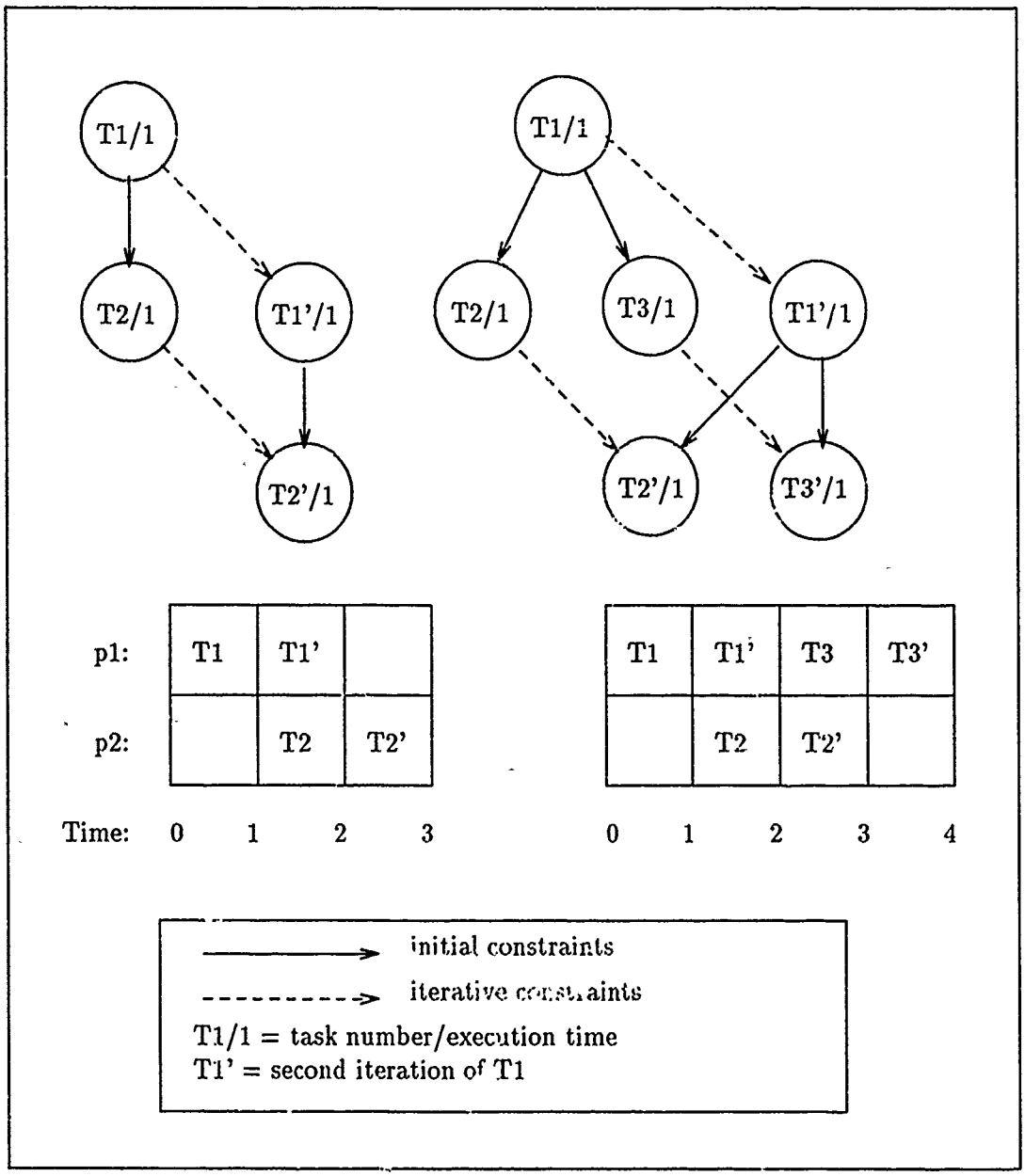



Figure 1.2. Task system structure changing under dynamic influences.

on the same processor. This condition can be a result of the scheduling process if interprocessor communications aren't considered when tasks are assigned to processors.

*1.3.3 Task Assignment Strategy* In order to optimally map a task system onto a parallel computer, the solution space of all possible mappings must be examined. One approach is the *greedy* method: a polynomial-time technique designed to generate a solution quickly. In this method, the mapping solution is generated step-by-step based on a set of candidates, a selection function, and an objective function. However, optimal results aren't guaranteed except in some restricted cases [8]. If more informed search techniques are used, optimal solutions can be guaranteed. Such techniques are in the class of '*best-first*' algorithms which can generate optimal solutions using admissible heuristic information [31]. Unfortunately, many require a prohibitive amount of physical system memory bases on the scheduling problem size.

## *1.4 Assumptions*

Before definitive optimization of static and dynamic iterative task scheduling can occur, the task system being examined must be defined. As a minimum, the following must be known [8]:

- The number of tasks within the simulation system.
- The execution times for each task.
- The task precedence-constraint relationships including feedback.
- The number of processors being scheduled.
- The number of execution iterations required to be scheduled.
- The type of system under consideration; *i.e.*, static vs dynamic.

Several theorems are presented in this document which represent formal descriptions of task scheduling properties based on task system characteristics. These theorem are offered without proof. The mathematical development presented, graphical explanations, and theorem simplicity should be sufficient.

## *1.5 Scope*

The primary objective of this investigation is to design, analyze, and validate effective mapping algorithms which generate optimally minimum schedules in terms of overall execution times on a parallel computer. The order of investigation in terms of problem complexity proceeds as follows:

1. The task system contains no feedback loops, no iterations, and all tasks have equal execution time (EET) requirements.
2. Iterations are permitted.
3. The system contains variable execution time (VET) tasks.
4. Feedback is permitted.

Using informed search techniques, mapping strategies for assigning tasks to processors in an optimal manner are implemented. These strategies derive an optimally minimum schedule of static iterative task systems for execution on a message passing parallel computer using the same computer. The characteristics of the simulation system are utilized to reduce the implicit search space and find an optimal solution in minimum time.

### *1.6 Approach*

This research analyzes scheduling algorithms which generate optimal mappings of tasks to processors. These algorithms range from those applied to polynomial-time scheduling problems to $NP$-complete scheduling problems [8, 38], *i.e.*, the solution space grows exponentially at best with the number of tasks to schedule. Such algorithms include the ***depth-first, breadth-first, backtracking, and best-first*** search methods. The 'best' algorithm in terms of search efficiency is implemented on an *iPSC/2* hypercube using the Ada programming language. Ada is chosen to investigate its utility on a parallel machine. Software engineering principles of modular design, strong intramodule cohesion and loose intermodule coupling are used to ensure a functionally sound implementation. Functional design techniques are used to transform the developed algorithm into an operation program. The resulting search performance given a range of task systems is analyzed for speed-up performance and scheduling efficiency. The programming code used to develop the parallel implementation is modified to improve the search process and generate optimal schedules more efficiently; *i.e.*, local bounding information is made global and load balancing among the processors is performed. The resulting algorithm strategy is validated using generic and VHDL task system simulations.

### *1.7 Summary*

This chapter discusses the advantages of parallel processing in producing solutions to complex problems much faster than with conventional computers. Although Amdalh's law appears to place a barrier on the potential speed-up which can be obtained when using a parallel machine, such serial

limitations approach zero when the problem size is scaled to fully utilize the available processing power of the parallel machine. Also, when using a parallel machine to execute VHDL simulations, the mapping of tasks to processors is crucial to the run-time performance. Therefore, generating optimal schedules whose schedule lengths are optimally minimum is essential in achieving the minimum run-time performance of the simulations. The scope of investigation presented starts with a simple task system and proceeds to the most complex VHDL simulation systems presently designed for parallel implementation; *i.e.*, VET iterative systems with feedback. The approach to designing an effective parallel algorithm for solving these complex problems analyzes many well-know search techniques for developing the 'best' serial algorithm. This serial algorithm is then parallelized for optimal search performance to produce optimally minimum parallel schedules of VHDL simulations for execution on a parallel, message passing machine.

### *1.8 Thesis Overview*

Chapter 2 is a background investigation of the generalized scheduling problem. The scheduling environment, policies, and goals are discussed to reflect the diverse nature of this problem. The theory of $NP$-completeness is introduced to show the complexity of scheduling a VET iterative task system. Also, many existing algorithms, their characteristics, and the multiple instruction, multiple data (MIMD) parallel architecture are discussed. Chapter 3 is the high level design of the algorithmic solution to the scheduling problems reflected in parallel simulations. The design issues such as machine memory limitations and search time performance are analyzed to determine the 'best' approach for implementation. In Chapter 4, the characteristics of simulation systems and three parallel search implementations are presented to identify the most efficient parallel implementation. Use of the Ada programming language is shown to be beneficial for this 'high-level' problem using the structure design methodology. Chapter 5 discusses simulation applications and the performance of the developed parallel algorithms in solving such parallel simulations. Chapter 6 reveals some important conclusions based on emperical analysis of the simulations used to validate the parallel algorithms.

# *II. Background*

## *2.1 Introduction*

An understanding of the iterative task scheduling problem, $NP$-completeness of the generalized scheduling problem, and parallel computer architectures and scheduling algorithms is essential to the completion of this research [2] [8] [14] [15] [18] [20] [21] [24] [35] [37] [39] [38]. Each subject area has been extensively explored by other authors, and many references are available. However, the focus of this chapter is to conduct a review of the reference material which is most applicable to the specific problem of iterative feedback task scheduling in a parallel environment.

The chapter is divided into five additional sections. Section 2.2 is an introduction of the general task scheduling problem with its diverse nature. Section 2.3 discusses the environment, the policies, and the goals of scheduling. Section 2.4 is an introduction to the theory of $NP$-completeness followed by a description of the $NP$-complete nature of certain task scheduling problems. Section 2.5 discusses the multiple instruction/multiple data (MIMD) architecture and some of the many scheduling algorithms and their characteristics under dynamic and static task systems. Section 2.6 is a summary of this chapter's contents. Within the text of this research discussion, task and process shall have the same meaning, as well as, concurrent and parallel.

## *2.2 Task Scheduling*

*2.2.1 The Task System Defined* A general task system can be defined using $\mathcal{F}$, $\prec$, $[\tau_{ij}]$, $w_j$ as follows [8:5]:

1. $\mathcal{F} = T_1, \ldots, T_n$ is a set of tasks to be executed.
2. $\prec$ is an (irreflexive) partial order defined on $\mathcal{F}$ which specifies operational precedence constraints, *i.e.*, $T_i \prec T_j$ signifies that $T_i$ must be completed before $T_j$ can begin.
3. $[\tau_{ij}]$ is an $mXn$ matrix of execution times, where $\tau_{ij} > 0$ is the time required to execute $T_j$, $1 \leq j \leq n$, on processor $P_i, 1 \leq j \leq m$.
4. The weights $w_i, 1 \leq i \leq n$, are considered deferral costs, *i.e.*, the cost of finishing $T_i$ at time $t$. Such a cost is simply $w_i t$.

When task iterations are included, the set $\mathcal{F}$ can be viewed as containing sets of tasks $T_1^1, \ldots, T_1^i$, $T_2^1, \ldots, T_2^i, \ldots, T_n^1, \ldots, T_n^i$ where $i$ represents the task iterations.

Graphical methods are employed to make understanding of such relationships clear [7]. The partial order $\prec$ is conveniently represented as a directed, acyclic graph (or *dag*) with no transitive arcs. The notation $T_i/\tau_i$ is used for labeling vertices of the *dag* where $T_i$ refers to a specific task and $\tau_i$ refers to the task execution time as described above. Figure 2.1 is an example of important characteristics of a task system which exhibits the following notation and properties:

1. Acyclic.
2. No transitive edges: $(T_1, T_6)$ would be such an edge.
3. $T_1, T_2$, and $T_3$ are initial vertices; $T_9$, and $T_{10}$ are terminal vertices.
4. $T_7$ is a successor of $T_1, T_2, T_3, T_4, T_5$ but an immediate successor of only $T_4$, and $T_5$; $T_5$ is a predecessor of $T_7, T_8, T_9$, and $T_{10}$, but an immediate predecessor of only $T_7$, and $T_8$.

The *critical path* of a vertex $T$ is defined as the sum of the execution times associated with the vertices in a path from $T$ to a terminal vertex such that this sum is maximal. Table 2.1 shows the critical paths and their values for Figure 2.1.

Table 2.1. Critical path execution lengths from each vertex $T$.

| $T$ | Critical Paths | Path Execution Lengths |
|---|---|---|
| 1 | $T1, T4, T7, T9$ | 8 |
| 2 | $T2, T5, T8, T10$ | 10 |
| 3 | $T3, T5, T8, T10$ | 9 |
| 4 | $T4, T7, T9$ | 7 |
| 5 | $T5, T8, T10$ | 8 |
| 6 | $T6, T9$ | 3 |
| 7 | $T7, T9$ | 5 |
| 8 | $T8, T10$ | 7 |
| 9 | $T9$ | 2 |
| 10 | $T10$ | 1 |

*2.2.2 Iterative Task Scheduling* Figure 2.1 is an example of a single execution, non- periodic task system. Each task is executed only once. When all tasks have completed execution, the task system has completed with no more task executions occurring. For an iterative task schedule, however, each task executes numerous times while maintaining the precedence-constraint relationship throughout each execution cycle. Figure 2.2 shows how such a task system can be viewed using graph theory as before. When feedback is added between iterations as shown in Figure 2.3, the resulting iterative relationship changes. The precedence-constraint relationships must remain

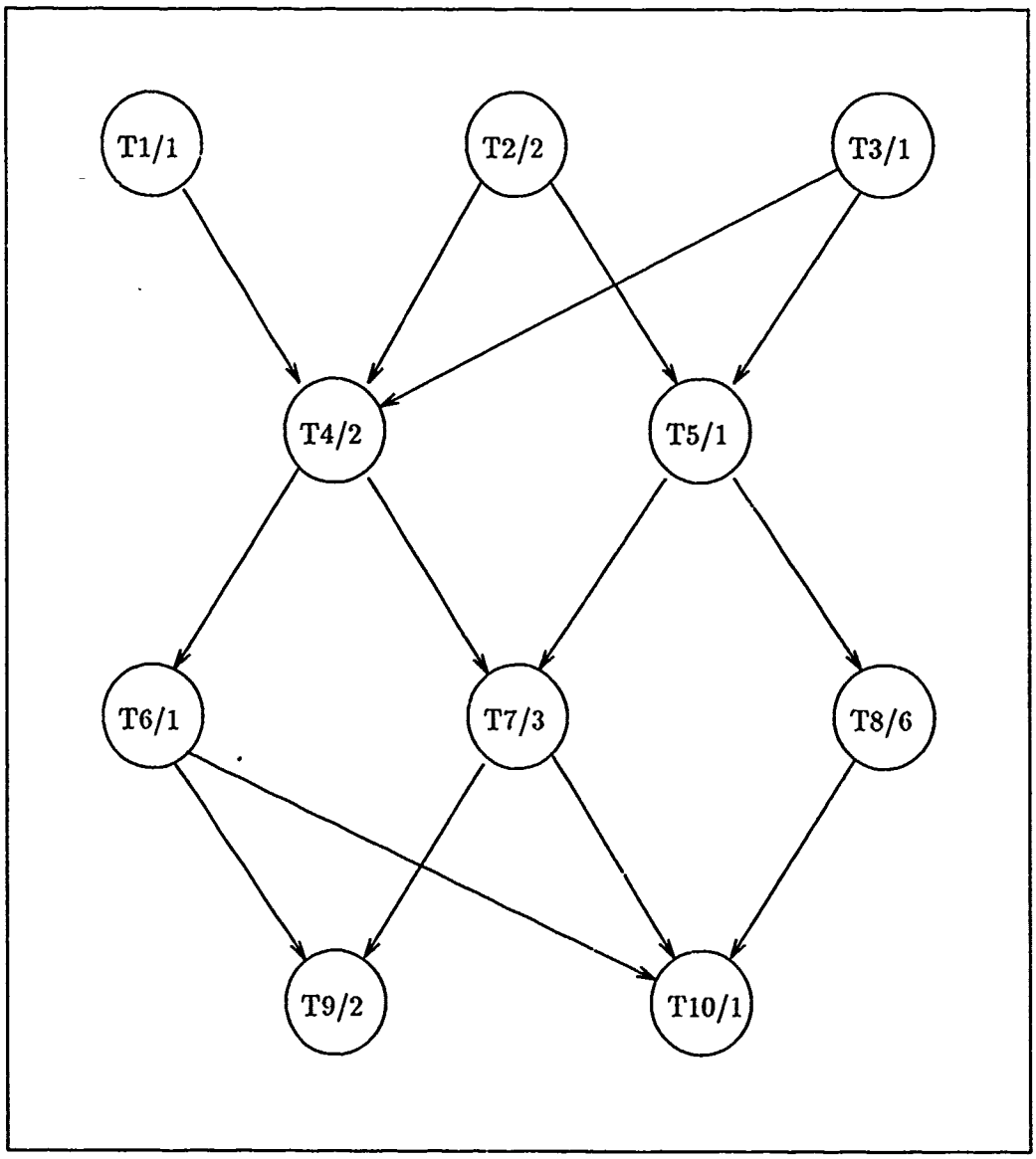



Figure 2.1. A dag representation of a task system

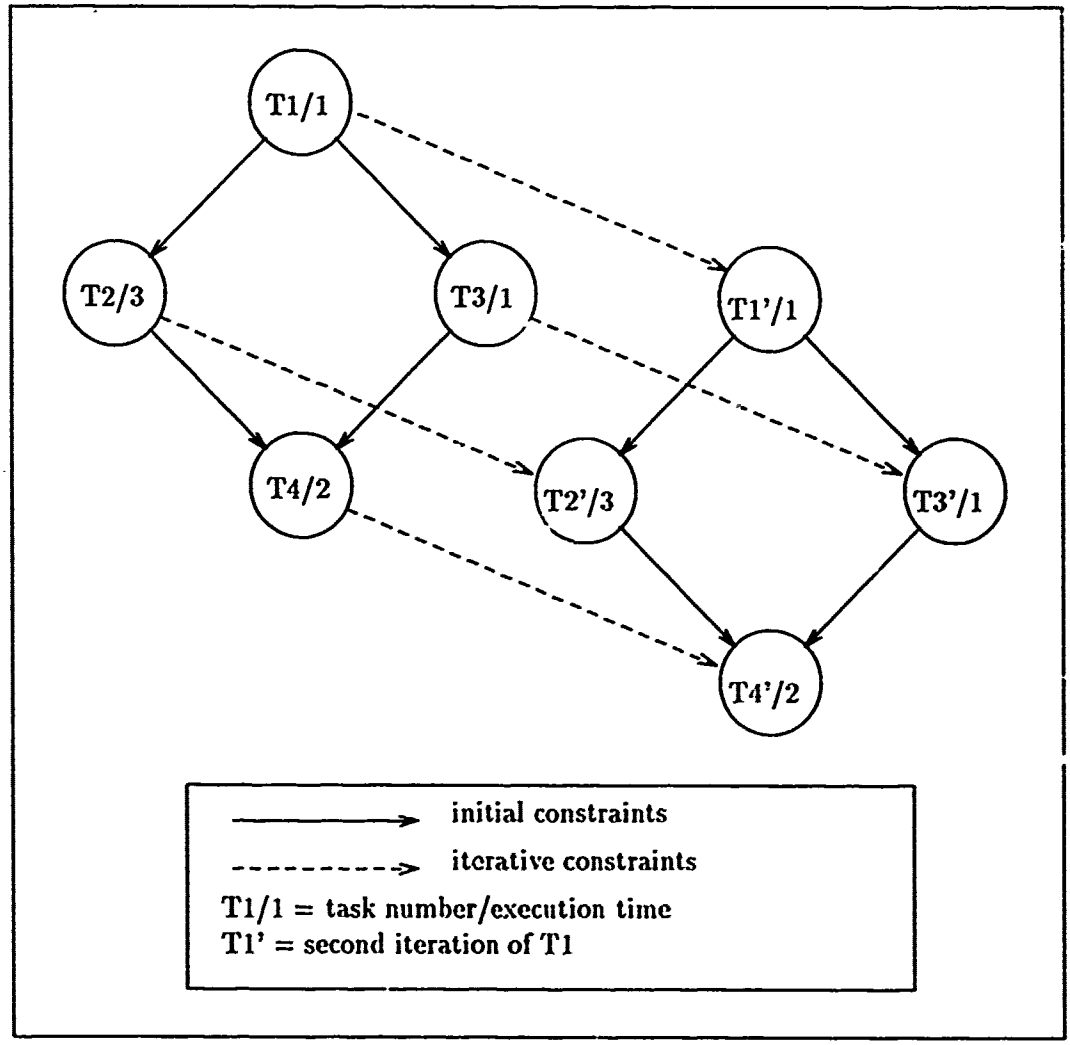



Figure 2.2. Representation of an iterative task system for 2 iterations.

intact which causes the graphical iterative representation to be adjusted as indicated in Figure 2.4. Although transient arcs can be introduced as Figure 2.4 shows ($T1'$ can be reach directly from $T1$ or through $T2$), they can be ignored since the feedback arc always introduces a new critical path from the task constrained by feedback ($T1$ in this case). For Figure 2.4, $T1'$ must now wait until $T2$ has completed regardless of when $T1$ completes. Therefore, the constraint between $T1$ and $T1'$ can effectively be ignored during the schedule generation process. The following theorems describe this change in the transitive closure of the original directed graph:

**Theorem 2.2.1** *Given a simulation system, the introduction of feedback dependencies removes from consideration all iterative dependencies of the task sending feedback information, the task receiving feedback information, and all tasks on all paths between these two.*

**Theorem 2.2.2** *Given a simulation system, the introduction of feedback does not increase the number of dependency arcs,* $\prec_{feedback} \leq \prec_{no\ feedback}$ .

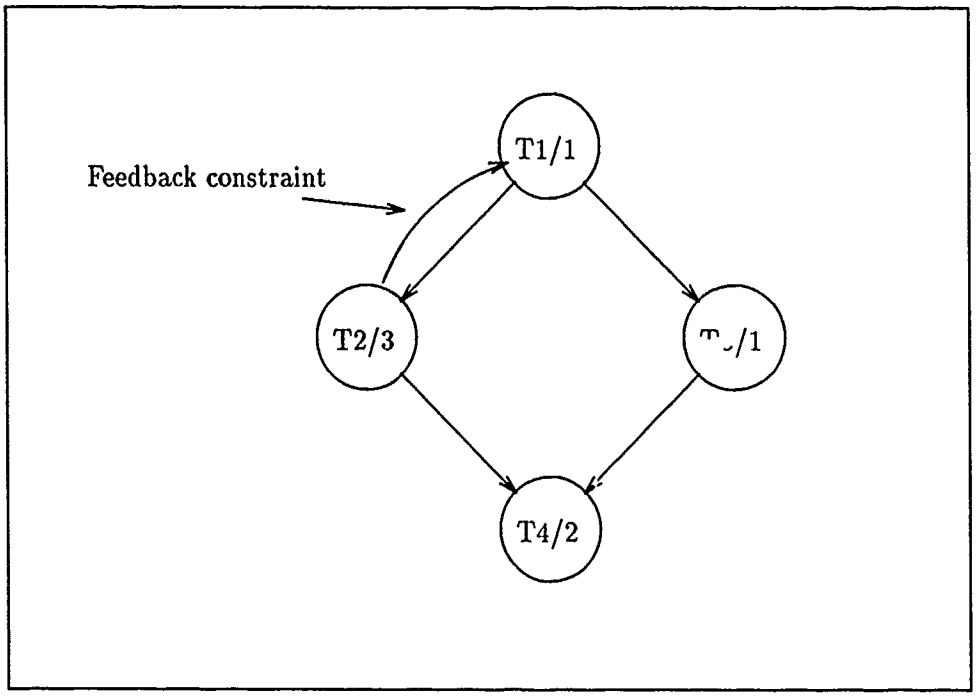



Figure 2.3. A task system with feedback.

Adding feedback to a given simulation structure can effectively reduce the number of scheduling paths through the iterative task graph.

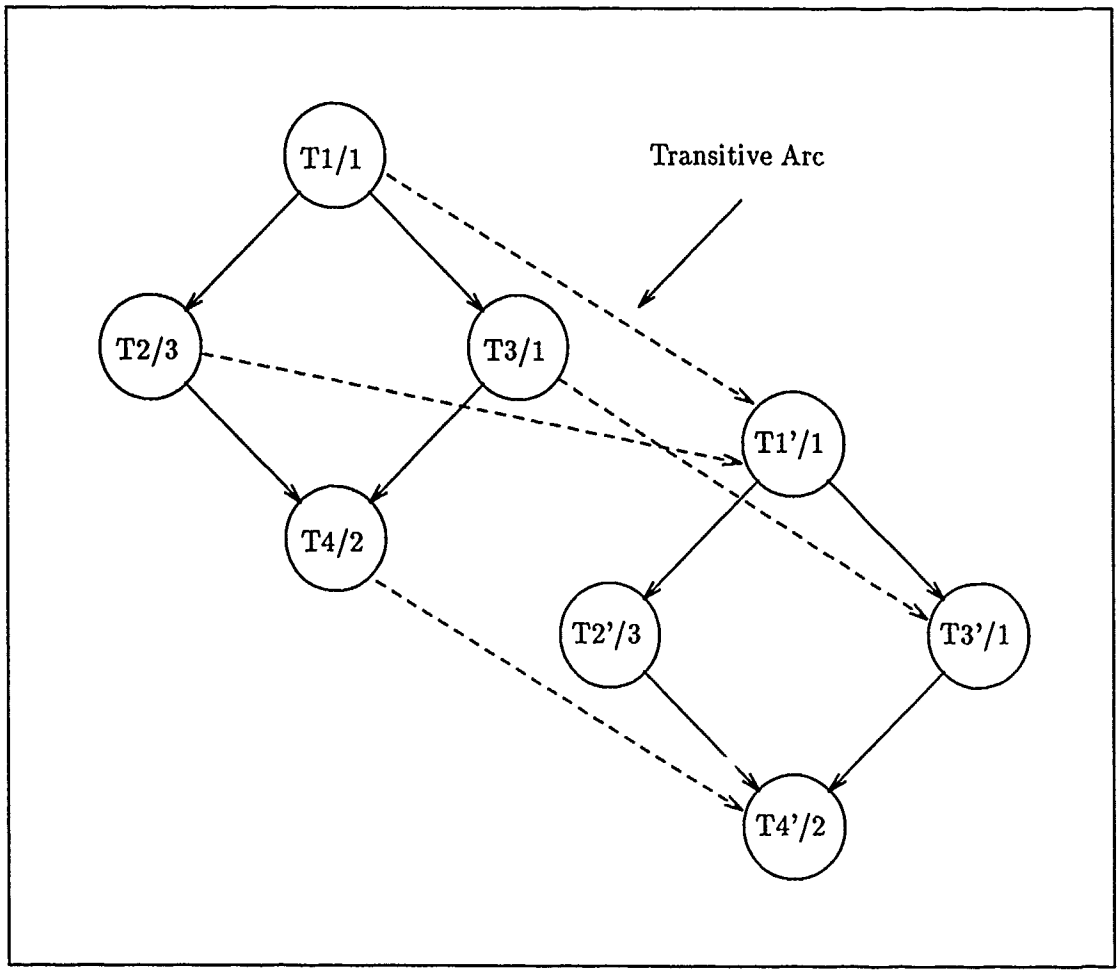



Figure 2.4. Representation of an iterative task system for 2 iterations with feedback.

*2.2.3 Performance Measures* Defining the requirements for the performance evaluation of parallel systems is essential [29]. Without firm understanding of specific requirements and the means to properly examine performance, much of the extracted performance evaluation data is useless. Two principle measures of schedule performance are the *schedule length* or *maximum finish* (or *flow) time* [8:9]:

$$\omega(S) = \max_{1 \le i \le n} \{f_i(S)\} \tag{2.1}$$

and the *mean weighted finishing* (or *flow) time*

$$\overline{\omega}(S) = \frac{1}{n} \sum_{i=1}^{n} w_i f_i(S) \tag{2.2}$$

where $f_i(S)$ represents the finish time for schedule $S$. For the general scheduling problems, therefore, efficient algorithms need to be found for the minimization of these quantities over all schedules $S$.

Performance measures for an iterative schedule where each task executes multiple times include an additional metric, latency. Latency is defined as the time between successive iterations of a given task [38:2-9]. For the iterative scheduling problems, therefore, efficient algorithms need to be found for the minimization of latency where multiple tasks are involved. The *level* strategy, which assigns tasks based on the longest chain of unscheduled tasks (critical path), forms a kernel for the iterative scheduling algorithm [39].

## *2.3 A Taxonomy of Scheduling*

[6] In order to fully understand the realm of task scheduling, the presentation of a taxonomy of scheduling problems is in order . The general scheduling problem can be viewed as consisting of three main components:

1. Consumers(s).
2. Resource(s).
3. Policy.

Understanding the functioning of a scheduler can best be done by observing the effect it has on its environment. In this case, one can observe how the *policy* affects the *resources* and the *consumers*. Such a relationship is shown in Figure 2.5. Figure 2.6 shows the structure of the hierarchical portion of the taxonomy of scheduling. A discussion of the relationship between the items at each level appropriate to the nature of iterative task scheduling follows:

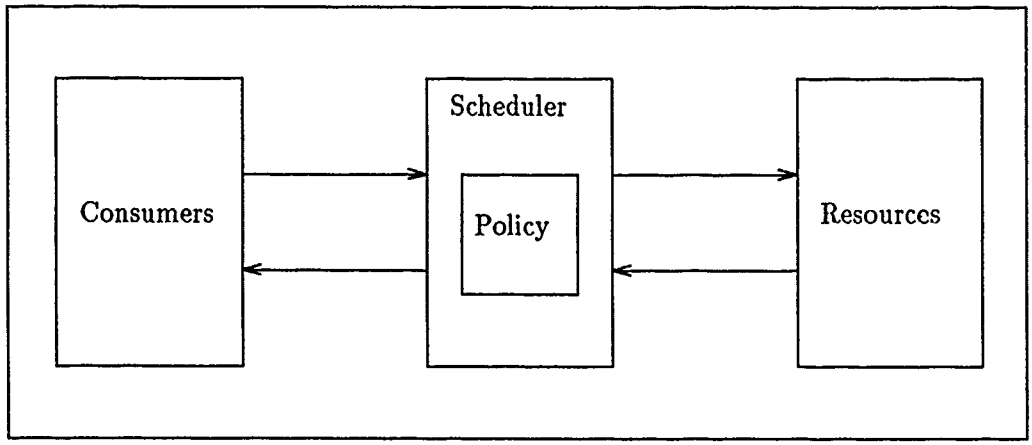



Figure 2.5. Scheduling system

- *Local Versus Global* Local scheduling involves the assignment of tasks to the time-slices of a single processor system. Global scheduling is the problem of deciding where to execute the tasks on a multi-processor system. In this case, a separate task scheduler is required to make these decisions rather than relying on the operating system of the single processor system.
- *Static Versus Dynamic* Static scheduling incorporates *a priori* knowledge of the task system to be scheduled. Assignments of specific tasks to specific processors are made prior to system execution. Dynamic scheduling, however, involves the more realistic assumption that very little *a priori* knowledge exits about the resource needs of a task prior to execution. In this case, it is the responsibility of the run-time system scheduler to make the appropriate decisions.
- *Distributed Versus Non-Distributed* The concern in this comparison is with the logical authority of the decision making process. Should the decision making authority under global dynamic scheduling reside with a single processor (*physically non-distributed*), or be distributed among the processors (*physically distributed*)?
- *Cooperative Versus Non-Cooperative* In a cooperative system, the processors cooperate between one-another in making scheduling decision. Each processor has the responsibility to carry out its own portion of the scheduling task, but with all processors working toward a common system-wide goal. In the non-cooperative case, the individual processors operate autonomously making scheduling decisions independent of the actions of the other processors. Such decisions are made regardless of the effects on the rest of the system.

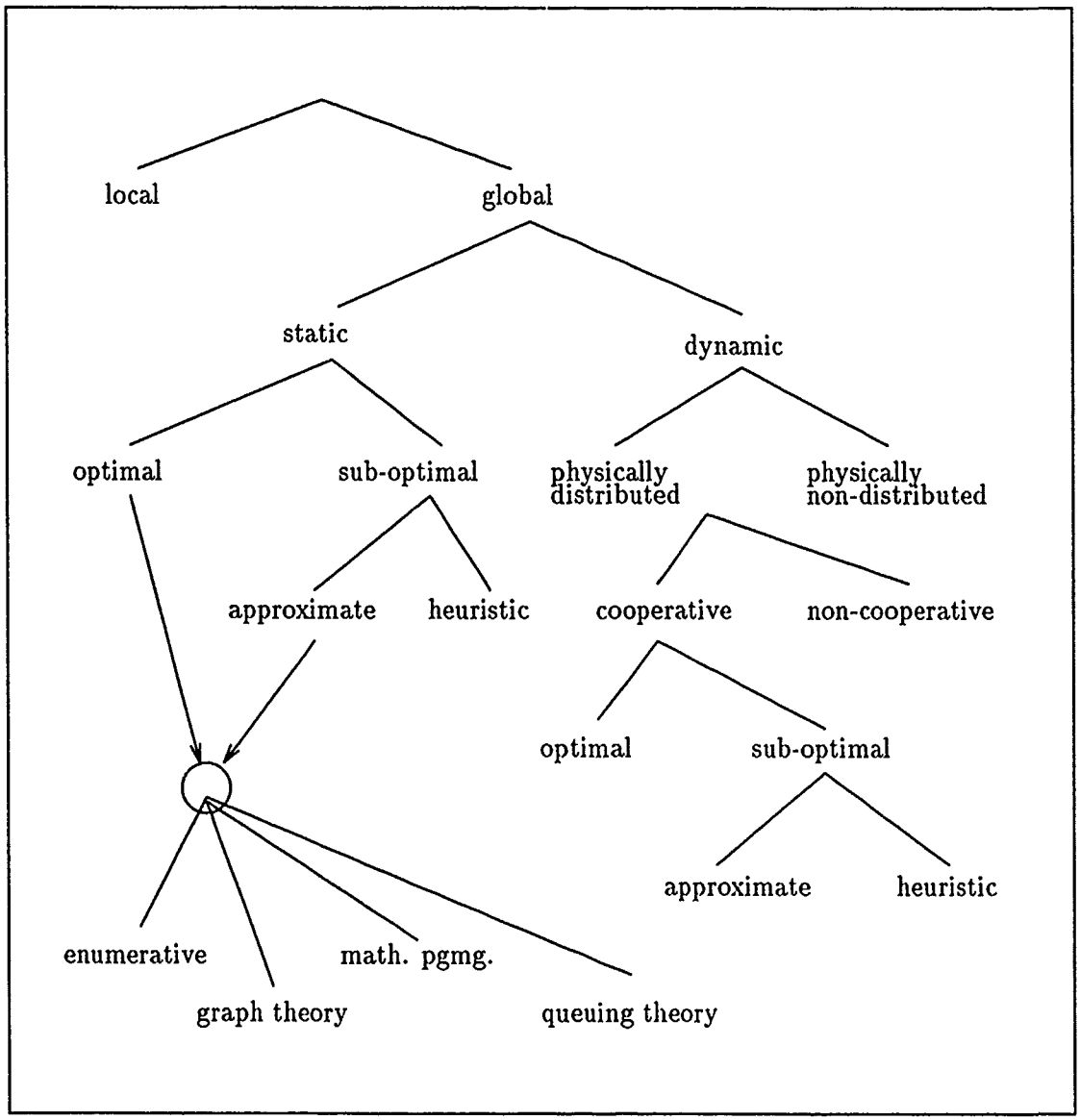



Figure 2.6. Task scheduling characteristics

- *Optimal Versus Sub-Optimal* Many specific optimal task scheduling problems have been shown to be $NP$-complete [8:20-21]. The dynamic iterative task scheduling problem being an extension of many of these problems exhibits the same characteristic. Therefore, optimal solutions (solutions which produce the best performance possible under a specific performance definition) are usually unfeasible in a run-time environment since the solution search time may be prohibitively costly. Sub-optimal solutions, however, may be produced without significantly degrading the system such that the cost of reduced performance is acceptable.
- *Approximate Versus Heuristic* An approximate solution is concerned with searching the solution space until a 'good' one is found instead of searching the entire solution space to find the optimal solution. Such a strategy is often referred to as the *greedy* approach. The time saving to generate this *good* solution can make it an acceptable solution (schedule). Unfortunately, determination of a *good* solution may not be insignificant, and the validity of this approach must be carefully analyzed. The heuristic solution, however, uses *a priori* knowledge concerning process and system loading characteristics to reduce the search space. Heuristic schedulers make use of special parameters which affect the system in indirect ways.

Policy decisions play a crucial role in the task scheduling mechanism. In the iterative dynamic task scheduling problem, the policy of *adaptive solutions* must be considered. An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to previous and current behavior of the system in response to previous decisions made by the scheduling system. The importance applied to the various parameters can vary from time to time depending if the scheduler believes certain parameters are providing information which is inconsistent with the rest of the inputs or are not providing any information regarding the change in system state in relation to the values of the others parameters being observed.

*Load balancing* is another important policy for multi-processor systems. The basic idea is to attempt to distribute the workload evenly among the available processors. The processors act together in order to redistribute some tasks from heavily loaded processors to lightly loaded processors. This policy relies on the assumption that the information at each processor is very accurate in order to prevent tasks from being endlessly circulated about between processors thus reducing overall system progress.

The policy of *bidding* is used in cooperative scheduling environments and involves all processors within the system. In this case, when a processor has a task awaiting execution, it announces the existence of this task and then receives *bids* from the other processors. Varied information can

be passed between processors to make the scheduling decision, and each processor maintains full autonomy.

Another classification of scheduling mechanisms is the *probabilistic scheduling* policy. Since the solution space for an optimal schedule can be very large, such a policy uses probability distribution information of the solution space to select a schedule. An important attribute can also be used to bias the random choosing process leading to a schedule better than one chosen entirely at random.

The nature of iterative task scheduling is complex and involves many strategies in obtaining a solution. These strategies all have unique characteristics which can be exploited for each scheduling problem. The determination of which policy to implement can be aided by an understanding of the environment in which the task system will be operating.

### *2.4 NP-completeness in Task Scheduling*

Many of the problems known and studied have solutions which can be found within two classes of computing times using the 'best' algorithms. The solution time for the first group of problems is bounded by a polynomial-time function; *i.e.*, there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size $n$ in $O(p(n))$ time. The second class of problems is those whose 'best' algorithms are nonpolynomial. Such problems are in the class $NP$-complete which can be defined by letting $\pi$ be a problem of size $n$ as follows [1:272]:

1. No sequential algorithm with polynomial running time is known for solving $\pi$ and, furthermore, it is not known whether such an algorithm exists.
2. All known sequential algorithms for solving $\pi$ have exponential running time and it is not known whether this is optimal.
3. If a solution to $\pi$ is given, it can be verified in polynomial time.
4. If a sequential polynomial time algorithm is found for solving $\pi$, it can be used to solve all $NP$-complete problems in polynomial time.

Unfortunately, efficient algorithms which produce optimal schedules and require only polynomial time are known only for a few task scheduling cases:

1. Scheduling on an arbitrary number of identical processors of an EET task system whose precedence-constraints form an in-forest or an out-forest (anti-forest) [8:54-59].

2. Scheduling of an arbitrary unit time task system on two identical processors [8:60-68].

3. Scheduling on an arbitrary number of identical processors of an EET task system whose incomparability graph is chordal [30].

This suggests that by restricting the precedence-constraints of a task system to certain subclasses which make the corresponding parallel programs more structured, other polynomial time algorithms may exist. However, as shown in [27], these subclasses exhibit the same complexity as scheduling EET task systems with arbitrary precedence constraints; *i.e.*, they are $NP$-complete. A more thorough list of scheduling complexities can be found in [8:20-21].

In general, for a system of $n$ independent tasks and $m$ processors, there are $m^n$ possible assignments of tasks to processors. If an optimal solution which minimizes the schedule length is desired, and the task system isn't structured as in one of the restricted cases above, then an extensive search process must be conducted. The classification of the iterative task scheduling problem is dependent upon the parameters used in the task system. As Table 2.2 indicates, the problem is $NP$-complete for variable execution time when the goal of minimum latency is desired [38:2-8]. Therefore, to solve the iterative scheduling problem for an optimally minimum schedule length results in exponential time requirements making the search prohibitively costly.

Table 2.2. Categories of iterative schedules for multiprocessors

| *Type* | *Parameters* | *Goal* | *Comments* |
|---|---|---|---|
| iterative | $n$ tasks | minimum latency | $O(n^3)$ if equal execution time |
| | $m$ processors | | $NP$-complete if variable execution time |
| | execution time $l_i$ | | |
| | $\prec$ | | |
| iterative | $n$ tasks | min # processors | $NP$-complete |
| | $m$ processors | | |
| | execution time $l_i$ | | |
| | $\prec$ | | |

### *2.5 Scheduling Algorithms and MIMD Machines*

The MIMD machine is a very powerful parallel processing computer with a wide variety of computer applications such as image processing and computer vision, artificial intelligence, operations research, robot arm control, and real-time high-speed simulations of dynamic systems to name just a few [20:1023-1029] [17]. Typically there are far fewer processors than elements to be processed, and so some natural aggregation of elements is required for partitioning the workload among the processors. For image processing, a spatial collection of elements or pixel data

is appropriate, whereas an aggregate of tasks which model the physical system is appropriate for computer simulations. Both the shared memory and the distributed memory versions offer distinct advantages when applications are tuned for the particular architecture. Those machines sharing a common memory are referred to as *multiprocessors* ( or *tightly coupled machines*) while those with an interconnection network are known as *multicomputers* (or *loosely coupled machines*). The shared memory machine doesn't suffer from the inherent communication delays of it's distributed memory counterpart; however, the application must deal with memory contention issues. The shared memory architecture has the advantage of easily accessible global data making it a powerful system when much communication is required between the processors within an application. However, the architecture is limited in its scalability due to bus contention issues. One such application known as Parallel Dynamic Interaction (PDI) [32] takes advantage of this architecture by maintaining limited global state information which all working processors routinely consult in order to make decisions in solving three $NP$-hard problems: flow-shop scheduling, job-shop scheduling, and vertex cover. Another application under process scheduling proposes a scheduling policy, deliberate random level-order scheduling with time slicing, and an implementation mechanism such that all processors can perform resource management in parallel and no central tables of resources need be accessed by processors running the resource management code [21].

Applications designed for the distributed memory MIMD computer must compensate for the increased communications costs between processors for effective machine utilization. Therefore, such applications must ensure the ratio of communications time to calculation time is kept small; *i.e.*, $\frac{T_{comm}}{T_{calc}} \ll 1$. This implies coarse-grain applications are best suited to such an architecture where this relationship is inherent within the control or data decomposition of the problem domain.

Many applications have been developed to examine the pros and cons of the loosely coopled machine. A comparison of three parallel $A^*$ search techniques: a *Shared-List* (centralized-list) algorithm which shares the search space among the processors, a *Static Distribution* algorithm which distributes the search space once to all processors (distributed list without load balancing), and the *Continuous Diffusion* algorithm where the search space is continuously redistributed demonstrates the inherent characteristics of this architecture. The results indicate that the Continuous Diffusion algorithm outperforms the others on the message passing architecture [12]. Also, comparison of the parallelized assignment problem using branch-and-bound techniques with load balancing supports the distributed-list approach (similar to continous diffusion) versus the shared-list approach when considering scalability, speed-up performance, and machine utilization efficiency [23]. Although asynchronous algorithms are difficult to design, evaluate, and implement, the concerns of parallel task scheduling such as assigning processes to processors in order to maximize system performance can be thoroughly addressed using the coarse-grain message passing architecture.

Within distributed computing, the central problem of task scheduling is motivated by issues such as load balancing, parallel algorithm requirements, algorithm-architecture matching, and utilization of resources [15]. Various algorithms have been developed to schedule task systems in static and dynamic environments each with unique policies and goals. Within the dynamic world, the goal of achieving flexibility through the dynamic scheduling of tasks in a distributed and adaptive manner has resulted in many such algorithms. K. Ramamritham and J. Stankovic describe (a) a locally executed guarantee algorithm for periodic and non-periodic tasks, which determines whether or not a task can be guaranteed to meet its real-time requirements, (b) a network-wide bidding algorithm suited to real-time constraints, (c) the criteria for preempting and executing a task so that it still meets its deadline, and (d) schemes for including different types of overheads, such as scheduling and communication overheads [35]. The algorithm was later extended to take into account precedence-constraints with cpu time being the only system resource explicitly taken into account. J. A. Stankovic and I. S. Sidhu describe a sophisticated and adaptive bidding algorithm for decentralized process scheduling in computer networks [41]. The algorithm is sophisticated because it attempts to match processes to processors based on many factors including process resource requirements, special resource needs, process priority, precedence constraints, the need for clustering and distributed groups (opposite of a cluster), specific features of heterogeneous hosts, and various other process and network characteristics. J. A. Stankovic also describes an application of Bayesian Decision Theory to the decentralized control of job scheduling [40]. In this paper, he presents a heuristic for the effective cooperation of multiple decentralized components of a job scheduling function. The heuristic he uses has an especially useful feature in that it can dynamically adapt to the quality of the state information being processed. S. Sahni develops other good heuristics to schedule tasks on computers that have multiple pipelined of multiple asynchronous processors [37]. C. Price analyzes software allocation models for distributed computing systems giving complexity results that describe the theoretical difficulty of obtaining exact and approximate solutions [34]. He examines an iterative transform, clustering, and banded Q heuristic algorithms and their relative performance against an optimal objective function. R. A. Beard and G. B. Lamont examine algorithm parallelism and performance improvements for the Set Covering Problem on a loosely coupled distributed parallel processor using coarse grain/static allocation, fine grain/dynamic allocation, and dynamic load balancing [3]. Another algorithm employed on a partitioned multiprocessor (PM) is the *Two-Tier Scheduler* (TTS) [14]. The PM has a shared global bus and nonshared local memories which allows local scheduling to amortize the cost of loading processes in local memory, and global scheduling to migrate processes for load balancing. This algorithm takes advantage of this particular architecture by adjusting a tunable time quantum so the average process completes execution on the processor on which it is first sheduled, and only relatively long lived processes are

rescheduled globally. A simple load balancing scheme for task allocation in shared memory parallel machines is described in [36]. The load balancing activity which ensures maximum utilization of available processing power is simple and distributed: whenever a processor accesses its local work pile of tasks, it performs a balancing operation with probability inversely proportional to the size of its workpile. When a balancing operation is performed, the work pile of a random processor is examined and tasks are exchanged so as to equalize the size of the two piles.

Classical task scheduling theory addresses only task sequencing. However, in [33], individual task parallelism is used such that applying more parallel processing power to the task allows it to execute faster. This theoretical analysis of generalized multiprocessor scheduling uses optimal control theory to solve the task scheduling optimization problem. All tasks are assumed to be dynamically partitionable, and the number of processors is treated as a continous variable permitting the application of the powerful techniques of continuous optimization to solve what would otherwize be a discrete problem. War gaming simulations can benefit from dynamic partitioning since the work load of the individual tasks during the simulation change due to the dynamics of activity levels within a battle field environment.

The task scheduling problem exhibits the same functional characteristics as the assignment problem: the assignment of processes to processors in order to optimize a system performance characteristic. When time critical, real-time operations are concerned, the search for optimal solutions can be too costly; *i.e.*, an optimal scheduling solution can be obtained only after some operational deadline. Such critical constraints can be found in research for the Strategic Defense Initiative. In [18], optimal and near optimal assignment algorithms using the technique of Marginal Assignment Potentials are coded and tested on a parallel processing system. The algorithms are iterative and interruptable, allowing assignment parameters to be updated and/or new targets added without having to restart the algorithm.

In static task scheduling, the application is relieved from the time constraints of dynamic scheduling; *i.e.*, previous scheduling decisions don't change during program execution. Unfortunately, many variables still must be considered in order to generate solutions whose schedule length is minimal. Many heuristic algorithms have been written to solve this problem. The *Earliest Ready Task* (ERT) algorithm considers communication delays between any pair of distinct processors and generates a makespan (schedule length) $M$ which always sastifies $M \leq (2 - \frac{1}{m})M' + C_{comm}$, where $M'$ is the optimal makespan without considering communication delay, $C_{comm}$ is the maximum communication delay in one chain, and $m$ is the number of processors being scheduled [24]. When a set of $n$ partially ordered tasks are given, the time complexity of this algorithm is $O(mn^2)$. A *Join Latest Predecessor* algorithm produces an optimal

schedule in linear time when there are enough processors to run all available tasks, and communication delays are no longer than the shortest task processing time [2]. Another algorithm which schedules tasks onto a partitionable mesh connected system (PMCS) uses a layer-by-layer partitioning strategy and a longest processing time first scheduling policy to generate the mesh size and the task assignment schedule [25]. The PMCS is a special VLSI device which must be attached to the host computer, and the host computer manages the PMCS. The algorithm accepts jobs in which their mesh requirements are known *a priori*. Two very powerful algorithms proposed in [19] solve very large scale problems of a few hundred tasks without regard to interprocessor communication delays. The *critical path/most immediate successors first* (CP/MISF) is an improved version of the CP-method or the highest levels first with estimated times (HLFET) method. The *depth first/implicit heuristic search* (DF/IHS) scheduling algorithm markedly reduces space complexity and average computation time by combining a lower bound function and the branch-and-bound method with CP/MISF.

### *2.6 Summary*

This chapter has focused on the iterative task scheduling problem, the inherent intractibility or $NP$-completeness of these problems, and the MIMD architecture upon which such problems can be effectively solved. Also discussed are the need for effective process schedules to make efficient use of these parallel machines, and many dynamic and static sheduling algorithms for ensuring machine utilization efficiency. The task scheduling problem is defined using graph theory as an acyclic directed graph. The iterative extension introduces transitive arcs which can be ignored from a scheduling point-of-view since the feedback constraints introduce new critical paths. Various system parameters are discussed to describe symbolically the iterative task system. The task system is then expanded to include periodic task execution requirements. A taxonomy of scheduling is also presented to show the diverse nature of task scheduling with the many policies upon which iterative scheduling can be based. The intractibility of the iterative task scheduling problem is also examined. Since the problem is generally $NP$-complete except when certain uniquely structured task systems are considered, the time requirement to generate and optimal solution is exponential. Therefore, the search for an optimal solution must consider this cost burden to determine the viability of such an approach. The MIMD computer architecture upon which such applications are well suited is also discussed. Various algorithms for scheduling static and dynamic task systems are reviewed, each unique in their policy drivers and the domain of performance criteria. Such algorithms are the basis for continued research in iterative task scheduling on MIMD computers. Since the objective of this research is the optimal solution of $NP$-complete iterative task scheduling, the *greedy* methods

(sub-optimal) and the *probabilistic* methods are not considered beyond their introduction in this chapter.

# *III. Scheduling Algorithm Design*

## *3.1 Introduction*

The generalized scheduling problem consisting of arbitrary precedence-constraints and variable execution time (VET) tasks is $NP$-complete as previously discussed. When tasks are allowed to iterate; *i.e.,* each task is permitted to repeat execution $i$ times, the solution space and time requirements increase dramatically if each task iteration is treated as a different task. The implementation of an efficient search algorithm to find and optimal solution which minimizes the schedule length for a distributed system on a parallel machine can result in near linear time speed-up making this an attractive approach in solving such a computationaly intensive problem. This chapter examines the algorithm design issues of generating such an optimal schedule using an Intel $iPSC/2$ hypercube. The scheduling policies and methods are discussed to give insight into the difficult nature of generating optimal schedules of a task system in a parallel environment. An optimal collection of the techniques completes the discussion.

## *3.2 Sequential Search Methods*

In searching for solutions to given problems, search graphs are generally formed which contain the search states explored at any point in time. A graph is normally produced because search states are often reached along more than one path from the initial state. However, maintaining a search graph structure can require a prohibitive amount of physical memory and slow the search process as well. One alternative is to create a search tree rather than a search graph where the control structure of the evolving search process creates a collection of search states connected in a manner much like the branches on a tree. The size of this structure can be kept within acceptable limits. Although this method creates the potential for evaluation of duplicate search paths, the diminished search performance may be acceptable. In the context of this discussion, search state and search node are synonymous.

Many well known search techniques exist for obtaining solutions to optimization problems [31]:

1. Depth-first – This strategy expands the nodes in order of increasing depth within the search tree. Each node chosen for exploration has all of its successors generated before one of these successors is chosen for future exploration. Once a solution is found, the search process is terminated. This technique works well when solutions are plentiful and equally desirable. When they are not, the search process can spend a considerable amount of time on fruitless

branches of the search tree. This strategy also must retain in storage the portion of the search graph that it is currently exploring.

2. Backtracking – This strategy is a version of depth-first search that applies the last-in-first-out policy to node generation instead of node expansion. When a node is selected for exploration, only one of its successors is generated, and unless it is found to be a solution or a dead end, it's again submitted for exploration. If the node meets some stopping criteria, the search process backtracks to the closest unexpanded ancestor; *i.e.*, an ancestor still having ungenerated successors. This policy doesn't suffer from any extensive storage requirements since only the current search path is retained.

3. Hillclimbing – This strategy repeatedly expands a node, inspects its newly generated successors, and chooses the best among these successors while retaining no further reference to the father or siblings within the search tree. This approach must retain in storage the current portion of the search graph only until the best successor is chosen.

4. Breadth-first – Unlike depth-first search which considers for future expansion only the nodes generated on the previous expansion, this strategy considers all successors at all levels within the search tree for possible exploration. This strategy guarantees to find the shallowest possible solution; however, the search process must retain in storage the entire portion of the search graph that it explores.

5. Best-first – The promise of a search node $n$ is estimated numerically by an heuristic evaluation function $f(n)$ which may depend on the description of $n$, the description of the goal, the information gathered by the search up to this point, and on any extra knowledge about the problem domain. The node with the lowest $f(n)$ is chosen for expansion. If two paths within the search tree lead to the same search node, the node with the higher $f(n)$ is discarded. This strategy also has the potential drawback in that the the search process must retain in storage the entire portion of the search graph that it explores.

6. Branch-and-bound – As the name implies, this strategy consists of two components: a branching process and a bounding process. The branching process always expands the search node most likely to be on the path to the desired solution. The bounding process eliminates from consideration those search nodes that can't lead to either a feasible solution or a solution better than one already found. An heuristic evaluation function is used for both purposes. Ideally, the search space is intelligently confined so that a minimal portion of the search graph must be retained in storage and the search process proceeds directly torwards a solution of the desired quality.

7. $A^*$ – This is a specialized best-first algorithm in whic!. the heuristic evaluation function $f(n)$ is defined as the sum of two components: $g(n)$ which the actual cost of the current search node, and $h(n)$ which is an estimate of the cost from the current search state to a goal state. With this additive evaluation function and delayed termination to prevent premature halting of the solution process when the first solution is found, this algorithm is guaranteed to find the optimal solution provided the heuristic evaluation function $h(n)$ is admissible; *i.e.*, $h(n) \leq h^*(n)\ \forall n$, where $h^*(n)$ represents the actual additional cost to a solution from the present search state. In other words, $h(n)$ is admissible if it never overestimates the actual cost to the goal.

8. ID$A^*$ [22] – Depth-first iterative-deepening is a stategy which compensates for the space requirements of breath-first search and the time requirements of depth-first search. The algorithm proceeds iteratively in a depth first search, cutting off a branch when its total cost ($g$+$h$) exceeds a given threshold. This threshold starts as the estimate of the cost of the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

### *3.3 Scheduling Combinatorics*

The search time for the generalized scheduling problem is exponential. At each point in the scheduling process, all possible combinations of task assignments must be implicitly generated in order to find the optimal solution. Figure 3.1 shows the scheduling combinations which exist given the intermediate schedule shown. At time $t_2$, both processors and three tasks are ready for scheduling. Therefore, three different combinations exist which must be generated for possible exploration. In general, given $n$ ready tasks and $m$ available processors, there are ${}_nC_m$ combinations of assignments to processors just for one expansion of a search node.

$$ {}_nC_m = \frac{n!}{m!(n-m)!} \tag{3.1} $$

This unfortunate consequence of the generalized scheduling problem leads to the generation of a number of successors which can exceed storage capabilities for all but the backtracking method described above. For instance, given a relatively small task system of less than 100 tasks to schedule onto 16 processors, if at one point in the search process, a search node is chosen for exploration which contains 50 ready tasks and 10 ready processors, $1.03 * 10^{10}$ possible successors can be generated. This space requirement for moving just one level down in the search tree is prohibitive on any machine.

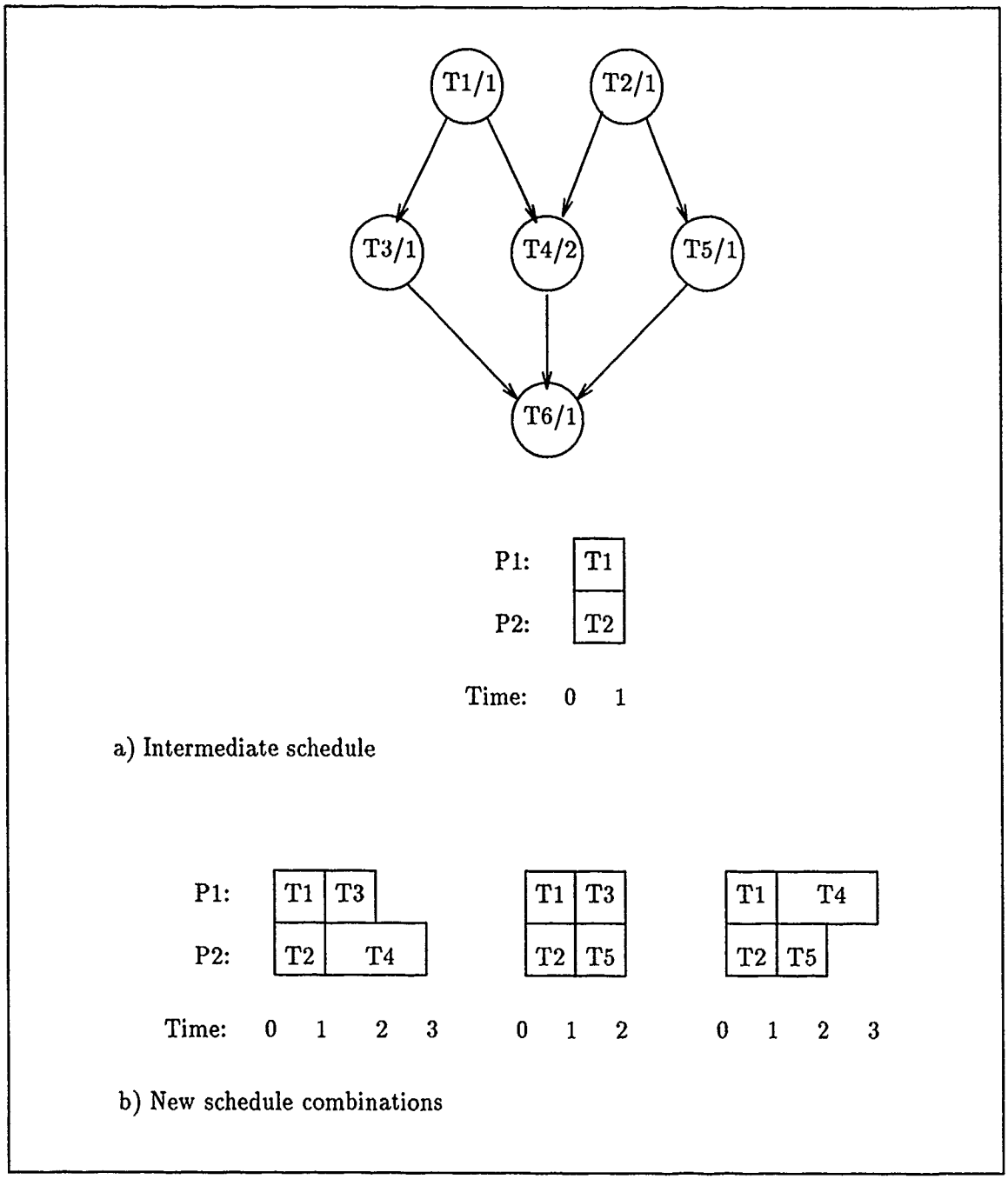



Figure 3.1. Scheduling combinations of a precedence-constrained task system.

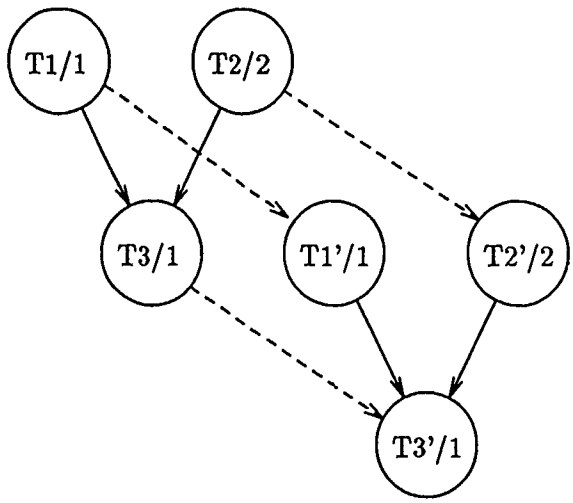
When task iterations are considered, their treatment in the scheduling process can increase the number of combinations of each search node expansion. For instance, if the *all-iterations-first* decision strategy for scheduling is maintained where once a scheduling decision for a task is made, all of that task's iterations are consecutively scheduled on the same processor at that time [38], then the problem reverts in numerical complexity to the single iteration problem. However, if each iteration is treated as a different task, then the number of successor combinations of any intermediate schedule in the search tree can be significantly larger. Figure 3.2 shows a simple task system where each task must execute twice. If the *all-iterations-first* decision strategy for scheduling is maintained, the search process generates only one successor (which happens to be the optimal solution in this case) from the intermediate search state shown. However, as Figure 3.3 indicates, if each iteration of a task is treated separately, then the search process must implicitly generate three successors of the intermediate search state to ensure an optimal solution is found. Therefore, treatment of an iterative task system in this manner has the effect of viewing a new task system with a number of tasks equal to the original number of tasks multiplied by the number of iterations thereby compounding the problem of limited storage capabilities.

### *3.4 Task Assignments*

The method by which tasks are assigned to processors is important in minimizing the search space and maintaining the search process on an optimal path. Since the focus of this research is in generating optimal solutions, the generation of all possible task assignments must implicitly occur. As shown in Section 3.3, the number of combinations of task assignments which must be implicitly explored can be prohibitive. However, if prudent selection of tasks is made, the number of combinations can be reduced significantly in some cases. A simple method for assigning ready tasks to ready processors is known as *List* scheduling. List scheduling is defined as follows:

- Whenever a processor becomes available, the list of tasks is scanned from left to right. The first unexecuted ready task encountered in the scan is assigned to the processor.

Unfortunately, many list scheduling anomalies exist [8:165-194] which can lengthen the schedule produced when (1) tasks are removed, (2) task execution times are all equally reduced, (3) precedence-constraints are weakened and, (4) the number of processors is increased. These anomalies can introduce serious doubt into the validity of an optimal schedule even when it is produced using a search algorithm such as $A^*$ which guarantees an optimal solution provided the additive evaluation function is admissible. The problem is not with the search algorithm itself, but with the list scheduling method by which the search states are created. As shown in Figure 3.4, removing




| | | |
|---|---|---|
| P1: | T1 | T1' |
| P2: | T2 | T2' |
| Time: | 0 | 1 2 |

a) Intermediate schedule where all iterations of T1 and T2 are assigned at $t_0$.

| | | | | |
|---|---|---|---|---|
| P1: | T1 | T1' | T3 | T3' |
| P2: | T2 | T2' | | |
| Time: | 0 | 1 | 2 3 | 4 |

b) New schedule combination where all iterations of T3 are assigned at $t_2$.

Figure 3.2. Search node expansion results for the *all-iterations-first* decision strategy.

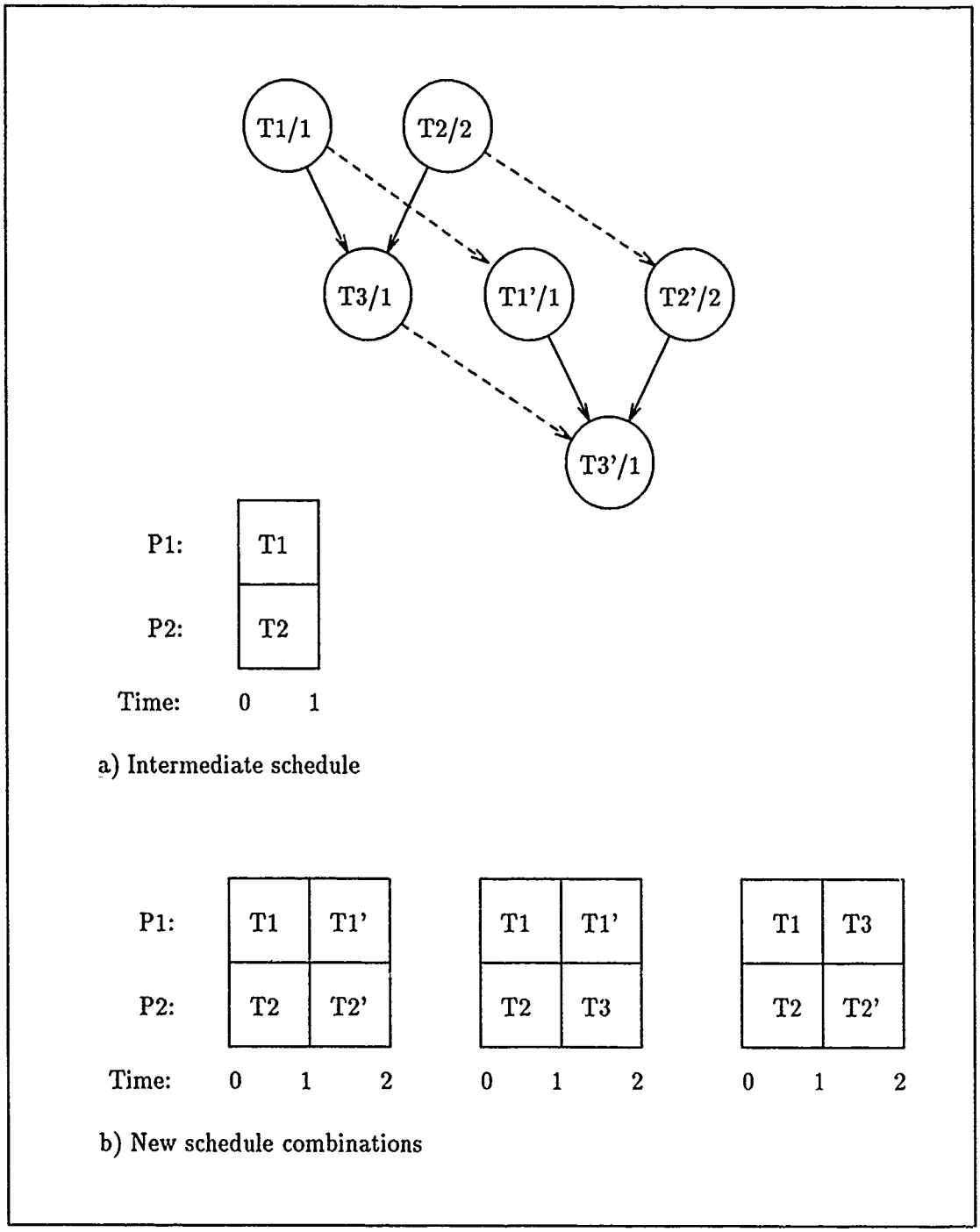



Figure 3.3. Search node expansion results when tasks iterations are treated separately.

a task from the system actually increases the schedule length. The search process is doomed from the start even with a guaranteed optimal search technique since the generation of the first search state under the list scheduling method will assign $T_1$ to $P_1$ and $T_5$ to $P_2$ at $t_0$ as shown.

It is easy to note in the previous example of a task system reduced by one task that if $T_5$ is forced to wait until $T_1$ has completed, the $A^*$ search technique using an underlying list scheduling algorithm will produce an optimal schedule. To compensate for the list scheduling anomalies when variable execution time tasks are considered, fictitious tasks which correspond to idle processors must be introduced. These idle tasks together with the ready tasks are then assigned to the processors as before. The length of these idle tasks must be $\min(\tau_i)_{i=1..n_{ready}}$ where $n_{ready}$ is the number of ready tasks being assigned. Let $m$ be the number of processors being scheduled, and $m_{av}$ be the number of available processors at a certain stage in the scheduling process. Then, the number of idle tasks $n_{idle}$ to be considered for assignment is [19]

$$n_{idle} = m_{av} - 1 \quad \text{for} \quad m_{av} = m$$

$$n_{idle} = m_{av} \quad \text{for} \quad 1 \leq m_{av} < m$$

Thus, the number of scheduling combinations generated at each expansion of a search node is given by

$$n_{branch} = {}_{(n_{ready}+n_{idle})}C_{m_{av}}$$

This shows that the number of successor search nodes created from a given search node can change dramatically if the guarantee of an optimal solution is to remain valid under list scheduling. Taking the previous example of 50 ready tasks and 10 available processors of a possible 16, an implicit generation of ${}_{60}C_{10}$ or more than $7 * 10^{10}$ successors is required. This is a seven-fold increase in the previous requirement. However, such compensation is only required for task systems with variable execution requirements. As shown in Figure 3.5, changing the execution times of each task so they are identical produces the same flow time when $T2$ is removed.

### *3.5 Parallel Search Methods*

Parallel machines play an important role in solving combinatoric problems which grow exponentially in space and time with the number of data items considered. Although physical limits exists as to the size of such machines in terms of the number of processors they can have, they still

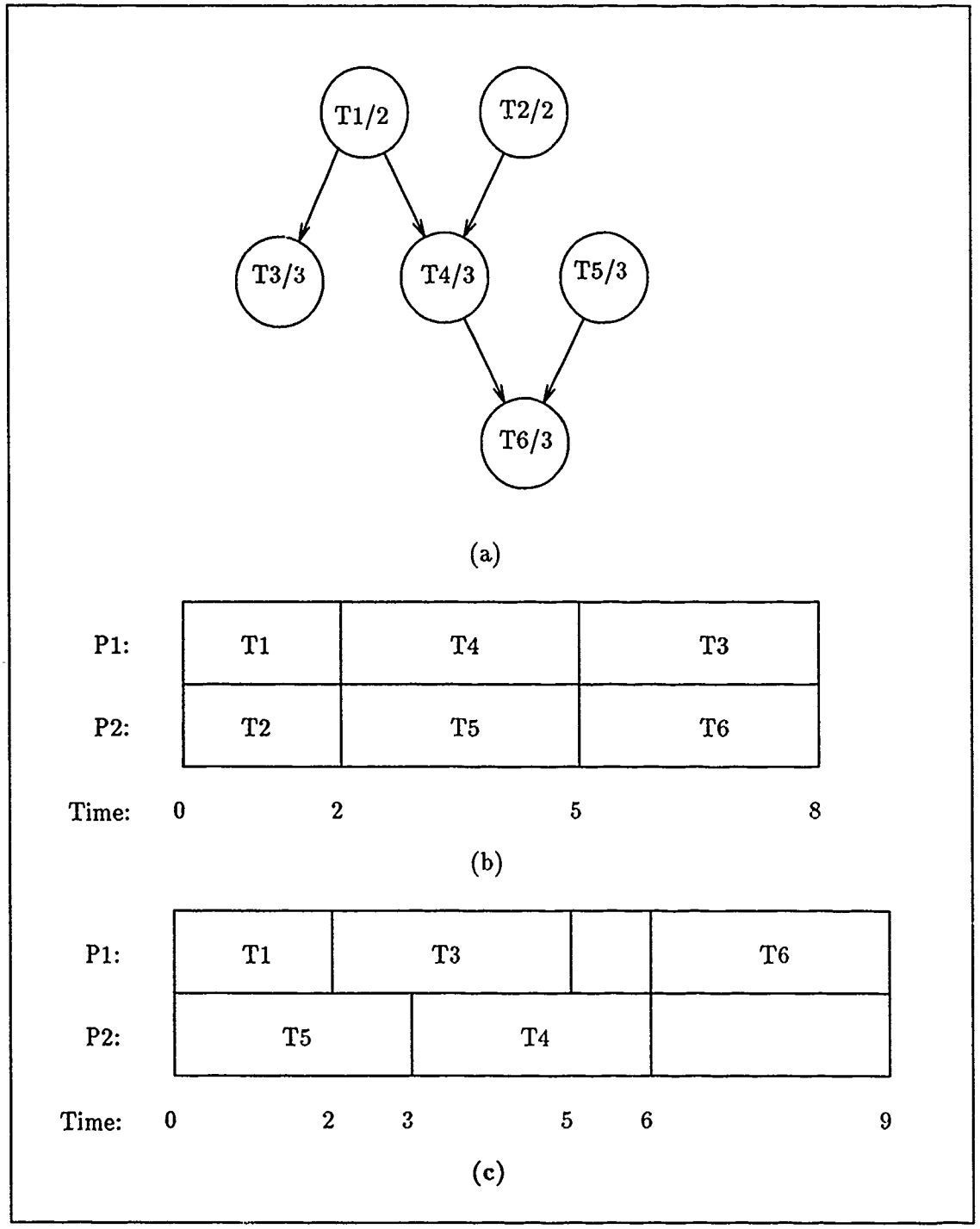



Figure 3.4. A list scheduling anomaly. (a) $\mathcal{F}$, $\prec$. (b) An optimal list schedule for $\mathcal{F}$, $\prec$. (c) An optimal list schedule for $\mathcal{F} - \{T_2\}, \prec - \{(T_2, T_4)\}$.

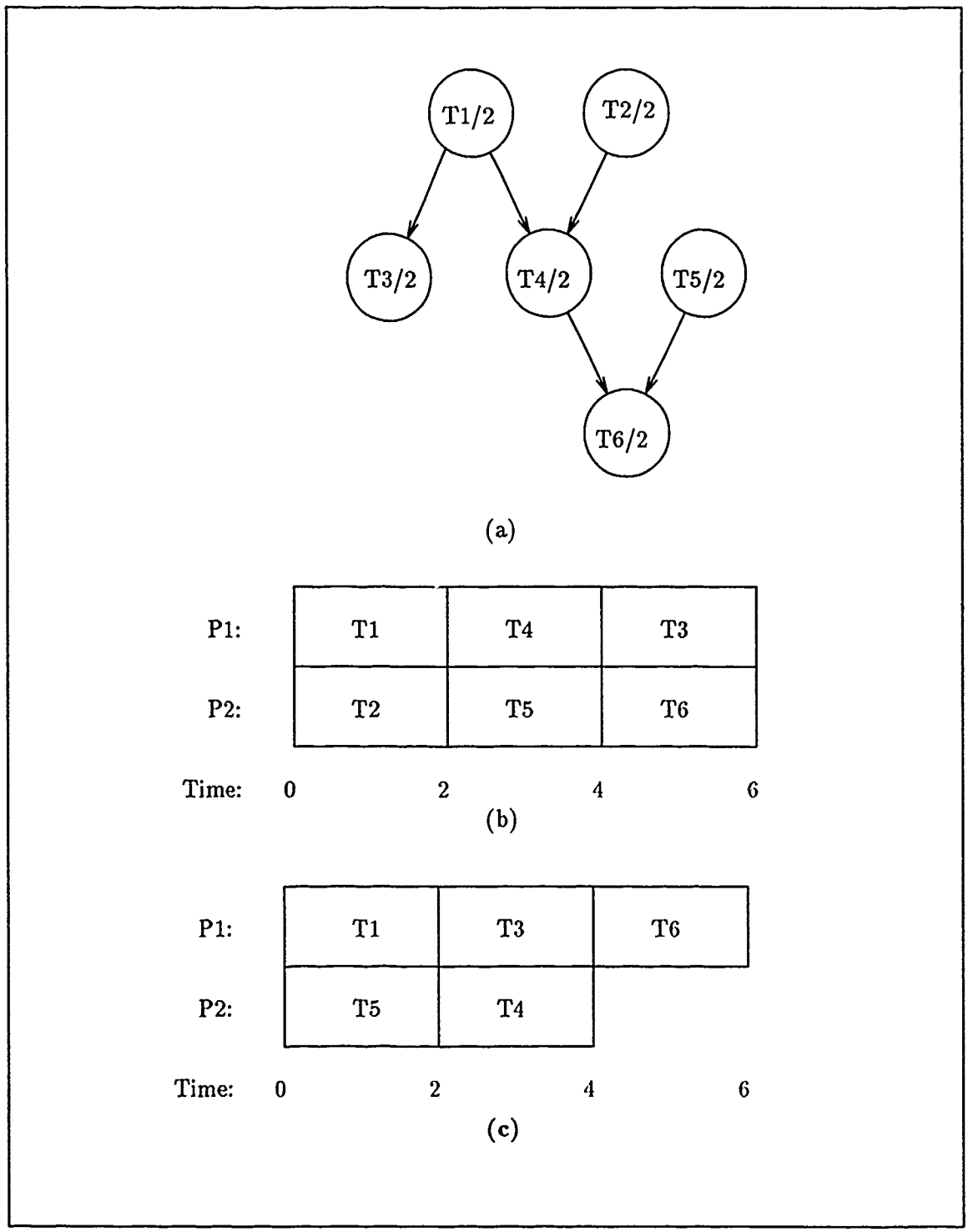



Figure 3.5. List scheduling anomaly disappears when task execution times are equal.

offer great improvements in complex problem solving abilities. Unfortunately, as the solution space of these combinatoric problems grows exponentially, the number of processors built into existing parallel computers grows linearly. Therefore, efficient algorithms are necessary to effectively utilize the processing power of these machines if the actual time requirements to achieve a solution are to be acceptable.

When considering the design of an efficient parallel algorithm, a determination of data versus control decomposition must be made. When decomposing a problem to be solved on a parallel computer under control decomposition, identifying and distributing unique control aspects of an algorithm allow each processor or unique collection of processors to perform different and essential functions in the execution of the algorithm. These processors work together on different aspects of the problem in trying to achieve improved performance over a single processor system. An example of control decomposition is a 15,000 line Fortran program which computes the electronic structure of high-temperature superconductors and other composite materials developed to run on a 128-node *iPSC/860* parallel supercomputer [4]. Alternately, data decomposition refers to the distribution of unique data items to all the processors. Each processor performs the same calulations on their unique sets of data. An example of data decomposition is the processing of pixel data in image processing where each processor is given a unique portion of the data set [17]. The data sets are all then processed in the same manner to produce the desired transformation. In parallel search techniques, the centralized-list approach is an example of control decomposition while the distributed-list approach reflects data decomposition of the problem domain.

In the algorithmic search process, data decomposition is the proper choice for machine scalability and efficient utilization [23] [12]. The limited number of different control functions within any of the sequential search techniques previously discussed prohibits control decomposition. A search graph contains numerous and identically structured elements which must all be processed in the same manner. Therefore, the distribution of portions of the search graph to different processors is the logical choice torwards achieving improved performance over the single processor system. However, without the processors having global knowledge of the local search graphs at each step in the generation of search nodes, the global search graph degenerates into a combination graph/tree where the branches of the tree are the local search graphs.

Many parallel search techniques exists for solving complex and data intensive optimization problems such as searching for optimal solutions to $NP$-complete problems. When implimented on a message-passing architecture such as the *iPSC/2*, the method by which the algorithm coordinates the search activity is crucial if considerable performance improvements over a single processor system are to be realized:

- Centralized-list – This algorithm maintains a central list of the search space currently awaiting exploration. One processors acts as the manager distributing search states to the other processors for expansion. After a predeterminded number of expansions, all the processors send back their search states currently awaiting exploration to the central manager which places them on a priority queue. The $m$ best search states are then redistributed to the other processors for further exploration. The process repeats until the entire search space has been exhausted. The algorithm suffers from the bottle-neck effect when the working processors all try to send their search states to the central manager at the same time. When the machine is scaled up in size (more processors), this effect becomes more pronounced reducing the overall utilization efficiency [12].
- Distributed-list – This algorithm eliminates the bottle-neck effect of centralized-list by distributing a portion of the search graph to each processor at the start of the search process. The processors all search independently until the entire search space is exhausted. The algorithm suffers from poor load balancing since the initial distribution of search space may be uneven.
- Continuous-diffusion [12]– This algorithm continously redistributes the search space among all the processors. After each processor expands a certain number of nodes, they each exchange their best search nodes with their nearest neighbors. This allows near-optimal search nodes to continually move from the current global and local minima to processors that don't have optimal nodes. Load balancing is assured, and machine utilization is maximized. However, exchange frequency must be 'tuned' for the undertaken problem since excessive exchange communications can increase the search time.
- Distributed-list with Load Balancing – This algorithm has the advantage of the distributed list approach with the policy of load balancing to maximize machine utilization efficiency. This approach moves portions of the search graph upon request from processors with available work to those that are idle.

### *3.6 Optimal Collection of Techniques*

The guiding factor in calculating an optimal solution to the iterative task scheduling problem is the combinatoric explosion of each search state expansion. For relatively small task systems where the number of potential search states awaiting exploration is not excessive, an informed search technique such as $A^*$ is very appropriate. Parallelizing this approach with the continuous diffusion technique produces the best combination of methods maximizing search efficiency. Unfortunately,

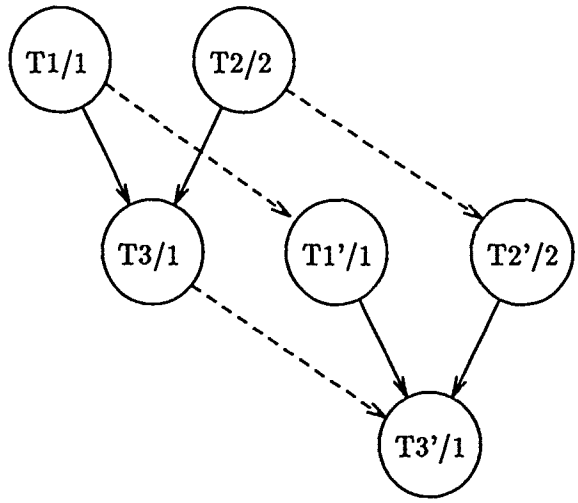
this method isn't scalable since machine storage capabilities can quickly be exceeded. The $A^*$ search technique is very good at keeping the search process focused toward an optimal solution while expanding the fewest search nodes of the serial methods previously described; however, it can't compensate for the combinatoric explosion in the generation of successor search nodes since each successor must be explicitly generated and maintained in system memory.

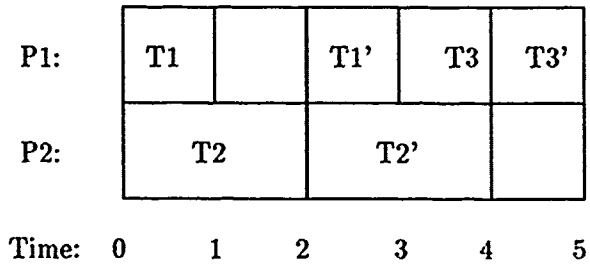
The only technique which doesn't suffer from this limitation is the backtracking method described in Section 3.2 where multiple successor nodes are implicitly generated. For efficient operation, the algorithm is modified to include an evaluation function as in $A^*$ with branch-and-bound decision criteria. The search technique is adapted to the distributed-list with load balancing method with the addition of interprocessor communications of upper bound information which emulates the continous-diffusion approach. This parallel modified backtracking (MBT) algorithm incorporates an early global termination check into the search process which checks the solutions against the initial lower bound flow time. If the search node is found to have a flow time which equals the initial lower bound, all processors are halted and the solution is reported. If the initial lower bound isn't met, but the searchnode is a solution and its flow time is better than any other solution previously found, this upper bound value is broadcast to all working processors to aid in the bounding process and reduce the enumerated search space. Once all processors have exhausted their search, the best global solution obtained is reported as the optimal solution. As figure 3.6 shows, the initial lower bound is based on the larger of the iterative task system's critical path, and $\lceil \frac{\sum_{i=1}^{n} k\tau_i}{m} \rceil$ where $k$ is the number of iterations, $\tau_i$ the execution cost of task $T_i$, $n$ is the number of tasks, and $m$ is the number of processors to be scheduled. Under no circumstances can a solution with a shorter schedule exist [38:4-28]. When feedback arcs are introduced, the initial lower bound can grow larger introducing more idle processor time given the same initial scheduling conditions. This result is shown in Figure 3.7.

### *3.7 Parallel Communications*

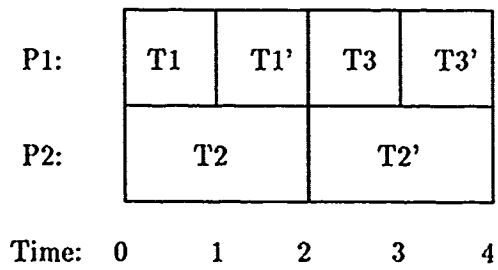
When designing an application to run on a coarse grain parallel computer such as the *iPSC/2* hypercube, two parameters are particularly important in characterizing machine performance:

- $t_{calc}$ – The time required to perform a floating point calculation.
- $t_{comm}$ – The time taken to communicate a single byte between two nodes.




a) Optimal schedule length = critical path




b) Optimal schedule length when $\prec = \emptyset$

Figure 3.6. Lower bounds on the schedule flow time.

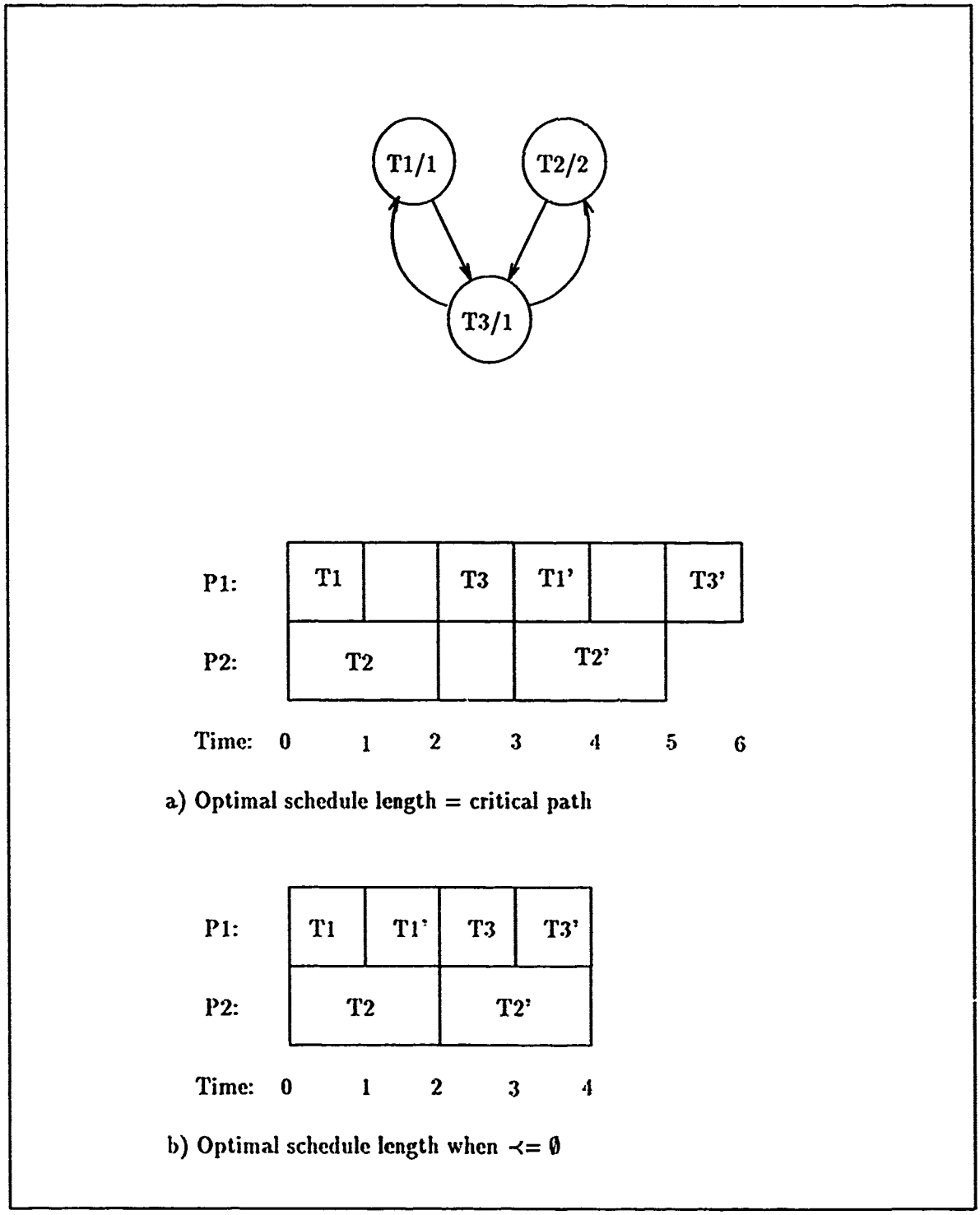



Figure 3.7. Lower bounds on the schedule flow time when feedback is introduced.

A relation which describes the granularity of a parallel machine and an application is the fractional communications overhead:

$$f_c = \frac{t_{comm}}{t_{calc}} \tag{3.2}$$

Ideally, $f_c$ is very small so that the full computing power of the parallel computer can be realized. Any time spent in communications resulting in idle processing time constitutes a penalty on the overall performance.

The *iPSC/2* hypercube is a coarse grain machine due to its message passing interprocessor communications topology. Therefore, $f_c$ is relatively large. In fact, given the communications throughput of 2.8 Mbits/sec, and a 80386 processor capable of 1 MFLOPS, the fractional communications overhead, $f_c$, is 35% considering the time to transmit one byte of information versus the time to perform one floating point calculation. The actual cost of communications and the effects of the underlying hypercube architecture is best understood when a normalized curve of $f_c$ is analyzed [28]. As shown in Figure 3.8, the cost of communications is highest when message sizes are small; *i.e.*, the number of floating point operations needed to equate to the time spent in transmitting a small message is high. This doesn't mean that large messages are desirable though. In fact, infrequent and small messages provide the best performance for a given application by allowing the processors to spend very little time idle.

### *3.8 Summary*

This chapter focuses directly on the scheduling process itself. Many of the serial search techniques for finding solutions to optimations problems are described to reveal some of their inherent limitations when dealing with potentially large search spaces. The backtracking method has the advantage of the implicit generation of multiple successor search states requiring a very limited storage capability, while the $IDA^*$ method combined minimum storage requirements with the guarantee of achieving an optimal solution given an admissible heuristic evaluation function. The combinatorics of the generalized scheduling problem are discussed to show the vast size of the potential search space and the prohibitive storage requirements for relatively small scheduling problems. The list scheduling method by which ready tasks are assigned to ready processors is examined to show the anomalies which can prevent the generation of an optimal schedule even though an $A^*$ search technique, which guarantees an optimal solution provided $h(n) \leq h^*(n)$, is used. Unfortunately, the solution to combat the anomaly aggravates the combinatoric explosion problem by introducing

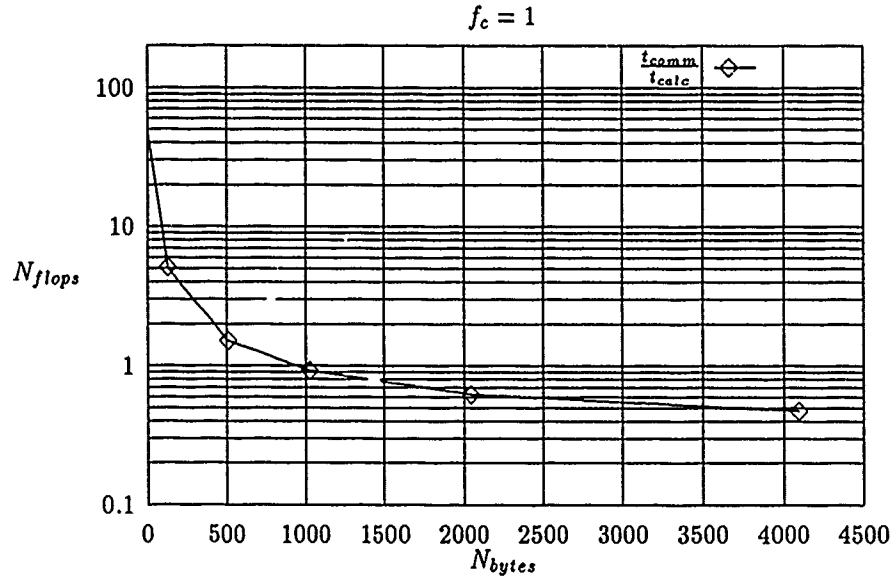



Figure 3.8. Normalized communications overhead of an $iPSC/2$ hypercube.

fictitious tasks to create idle processor times in the schedule generation process. Several parallel search techniques are also discussed to show the manner in which a data decomposed search problem can be solved on a parallel computer. Based on the search methods and implementation techniques for parallelizing the search process, the best combination of methods is presented. This parallel MBT algorithm uses a distributed-list with load balancing technique to maximize search efficiency providing the 'best' combination for obtaining a near linear speed-up of the search process on a parallel, message passing architecture while not exceeding the physical storage limits of the computer. Also, the communication characteristics of a coarse-grain computer such as the $iPSC/2$ hypercube is shown to require application decomposition to result in computationally intensive sub-problems with minimal communications to maximize the machine's processing potential, thus, keeping overall processing times minimal and machine utilization efficiency high.

# *IV. Low-Level Design/Analysis/Implementation*

## *4.1 Introduction*

The underlying search technique by which an optimal schedule is obtained determines the success or failure of such an enterprise. Machine memory capacity and search time must be considered because of physical limitations of hardware and human patience. This chapter examines the requirement for a modified backtracking (MBT) search technique and its characteristics. Also studied are the data characteristics of simulation programs whose optimal execution schedule for a message passing architecture such as the *iPSC/2* is desired. Whether or not the task system contains forward precedence constraints, feedback precedence constraints, or equal task execution times can significantly effect the search processor. Based on *a priori* analysis of a task system's characteristics, several methods of generating schedules are conjectured to produce optimal schedules in the most efficient manner. Three parallel techniques for distributing the search effort using the distributed-list approach are discussed to reveal unique characteristics of the search tree in generating optimal schedules. The use of Ada in the development of programs for validating these schemes is also shown to be beneficial using the structured design methodology.

## *4.2 Backtracking Search Variations*

As described in Chapter 3, the backtracking (BT) search technique requires the least storage requirements of any search technique since only search nodes on the current search path need to be maintained. Since all the search nodes generated are discarded except those on the current search path, the search control structure evolves as a tree instead of a graph. When the algorithm includes delayed termination where the entire search tree is explored, an optimal solution can be obtained [31]. When bounding information is included in the search process, the search tree can be trimmed to reduce the number of search nodes, and thus, the search time. The current best solution value obtained throughout the search process is used to bound the global search tree and reduce the number of search nodes which must be generated and evaluated. When heuristic information (decision rules) is included in the search process, even more search nodes can be eliminated from the search tree (branch-and-bound) providing a greater reduction in search time. This modified backtracking process can significantly reduce the size of the search space as shown in Figure 4.1. Part (a) represents the complete enumeration of the search space. Part (b) shows how using the current best solution value can reduce the search space by reducing the number of search nodes which must be generated. Part (c) shows how the search space can be reduced in size even further when heuristic information is used.

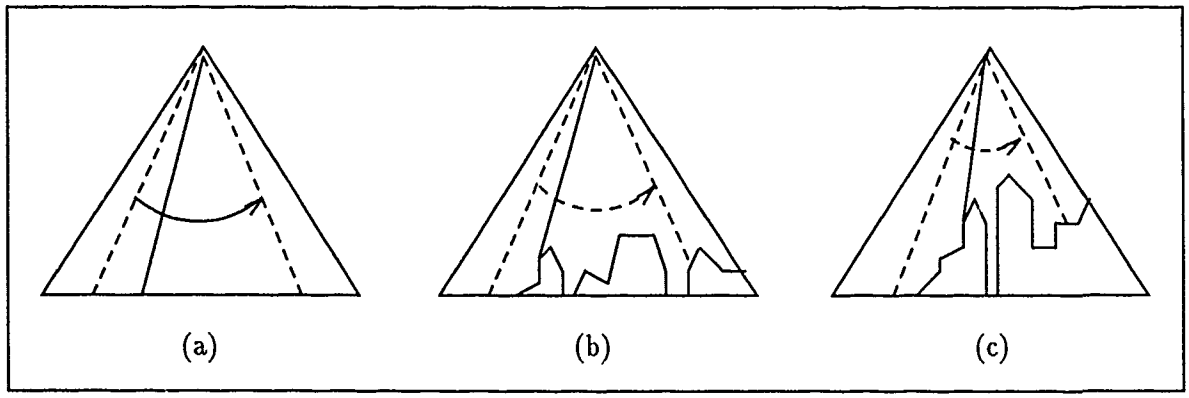



Figure 4.1. Schematic representation of three BT strategies.

*4.2.1 Combination Trees* When analyzing a balanced, non-uniform combination tree where the branching factor at each level of the tree can be different as shown in Figure 4.2, the number of states which exist can readily be found using the following equation:

$$States_{total} = 1 + (\sum_{i=1}^{L-1} \prod_{j=1}^{i} bf_j) \tag{4.1}$$

where $L$ is the number of levels in the tree. Adding 1 to the total accounts for the first state at the top of the tree. Note that this expression reduces to the familiar expression

$$States_{total} \quad bf^L - 1 \tag{4.2}$$

when the branching factor at each level is the same.

An upper bound for the size of the search space (total number of states), is easily found when relaxing the precedence-constraints of a task graph. Initially, there are ${}_nC_m$ ways of assigning the first set of tasks to processors. To ensure all solutions are obtained for VET task systems, the remaining sets of assignments must consider permutations. Therefore, there are ${}_{(n-m)}P_m$ ways of assigning the second set of tasks to processors, ${}_{(n-2m)}P_m$ ways of assigning the third set of tasks to processors, etc... The last set of task assignments must consider the case where the number of remaining tasks is less than the number of processors. In this case, the problem instance can be viewed as assigning processors to the remaining tasks. Therefore, the total number of states which

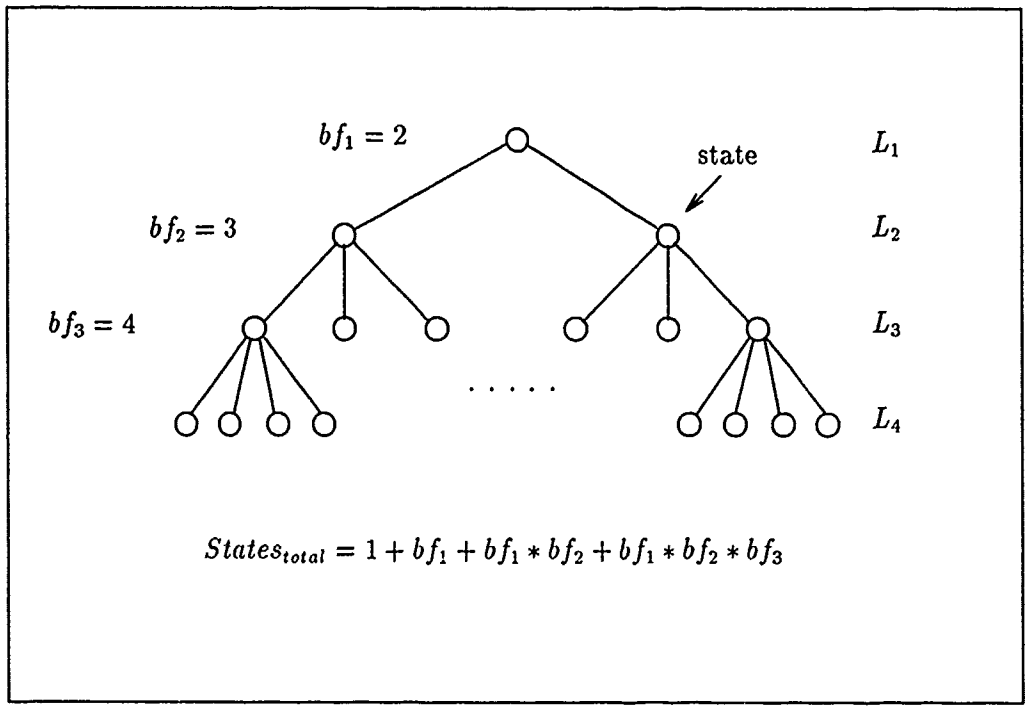



Figure 4.2. Non-uniform combination tree.

exist within the search space is found by setting the branching factor equal to the permutations of each task assignment set:

**Theorem 4.2.1** *$States_{total} = 1+(\sum_{i=1}^{L-1}\prod_{j=0}^{i} bf_j)$ where $bf_j = {}_nC_m$ for $j = 0$ and ${}_{(n-(j-1)m)}P_m$ for $j > 0$. For $nMODm \neq 0$, $bf_{L-1} = {}_{(m)}P_{(n-(L-2)m)}$*

When the task system contains EET tasks, the permutation term can be reduced to combinations. Also, if $nMODm \neq 0$, then the final term = 1:

**Theorem 4.2.2** *$States_{total} = 1+(\sum_{i=1}^{L-1}\prod_{j=1}^{i} bf_j)$ where $bf_j = {}_nC_m$ for $j = 0$ and ${}_{(n-(j-1)m)}C_m$ for $j > 0$. For $nMODm \neq 0$, $bf_{L-1} = 1$*

Unfortunately, the total number of search states is still rather large. When considering precedence-constraint task systems, the total number of search states is reduce since not all assignments are valid; however, the exact number of total search states for each task system is unique to that system and highly dependent upon its precedence-constraint structure. Therefore, the number of potential assignment combinations for each level when used as the branching factor overestimates the number of combinations since precedence-constraint relationships restrict many combinations from being realized; *i.e.*, the combination tree becomes narrower and deeper as shown in Figure 4.3. Also, when compensation for the list scheduling anomalies is applied to the scheduling process, the combination tree becomes unbalanced. This results from an uneven distribution of tasks within each level of the combination tree due to the idle processor times introduced as shown in Figure 4.4. Therefore, to ensure a true upper bound is obtained, the $n$ terms of Theorum 4.2.1 must be increased as outlined in Chapter 3.

*4.2.2 Search Tree Pruning* Better understanding of the effects of search tree pruning is found when Figures 4.5 and 4.6 are examined. This iterative task system contains only five tasks of equal execution time. Two iterations of each task are required to be scheduled onto two processors. A complete expansion of the search tree produces 10,225 search nodes when idle processor states are forced which compensates for the list scheduling anomaly! When upper and lower bound values are used, the enumerated search tree is dramatically reduced to 65 search nodes. The upper bound value is simply the current best solution value throughout the search, and the lower bound value is the initial calculation of the larger of two values: $\lceil \frac{\sum_{i=1}^{n} k\tau_i}{m} \rceil$ where $k$ is the number of iterations, $\tau_i$ the execution cost of task $T_i$, $n$ is the number of tasks, and $m$ is the number of processors to be scheduled, and the critical path value of the initial task graph including iterations. When heuristic information is used as well, the enumerated search tree is further reduced to 10 search nodes (the

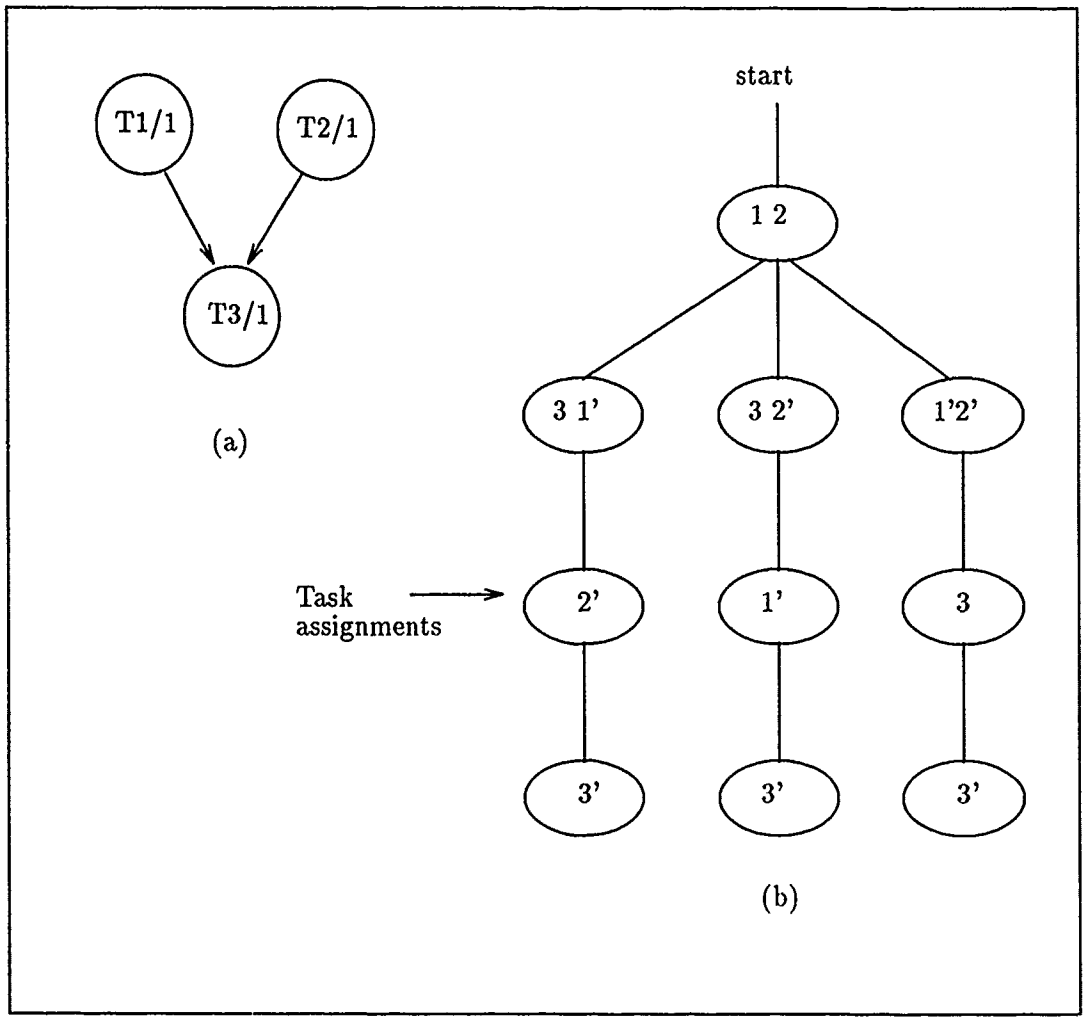



Figure 4.3. (a) Task system. (b) Combination tree of task assignments for 2 iterations without list scheduling anomaly compensation.

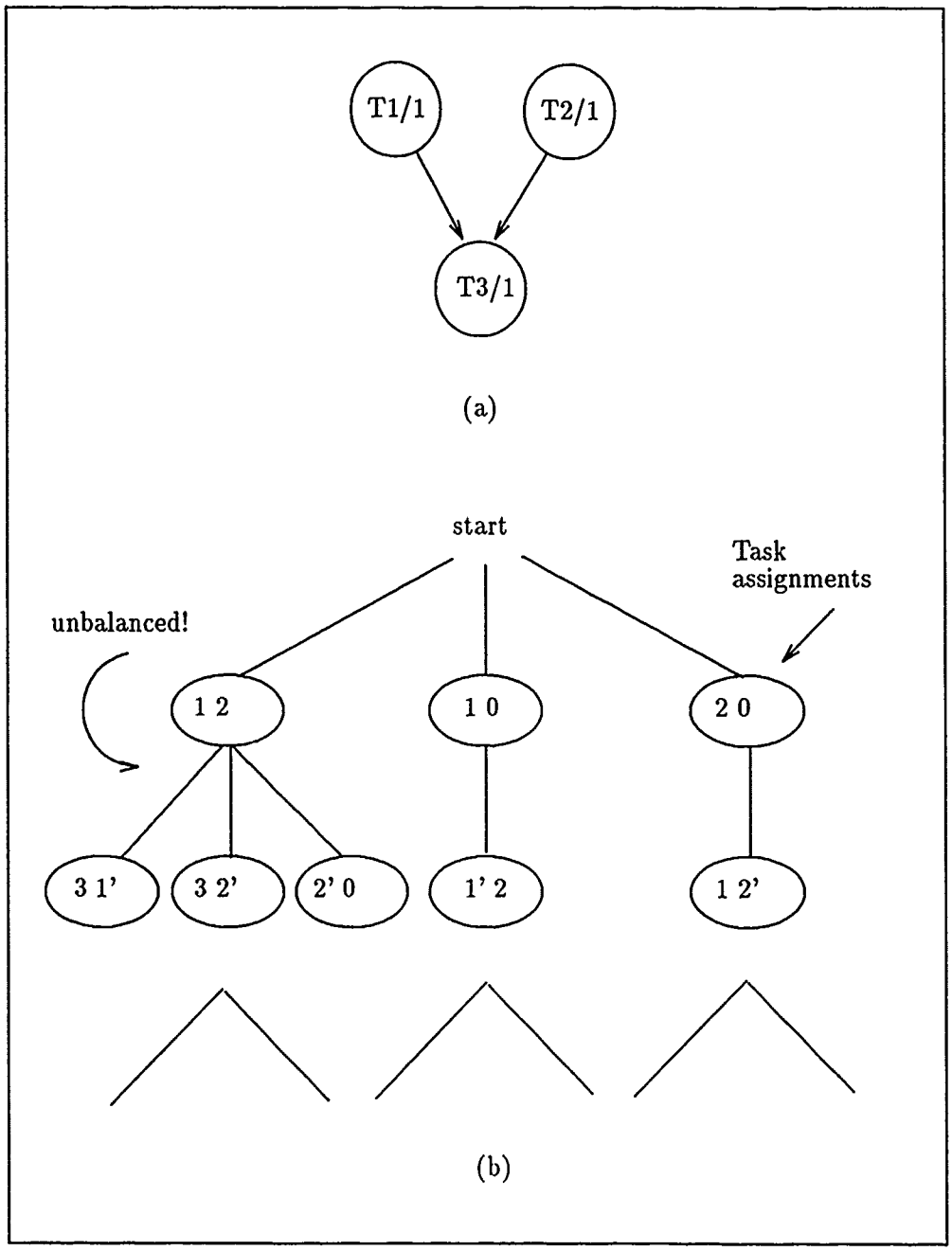



Figure 4.4. (a) Task system. (b) Combination tree of task assignments for 2 iterations with list scheduling anomaly compensation ('0' represents idle processors).

numbers above the search nodes in Figure 4.6 represent the order in which the search nodes are generated). The additive evaluation function $f(n)$, where $f(n) = g(n)+h(n)$, is calculated for each intermediate search state in the same manner as the initial lower bound value is calculated. For this particular task system, an improvement of four orders of magnitude is realized between fully enumerating the search tree and using heuristic information to reduce the search space.

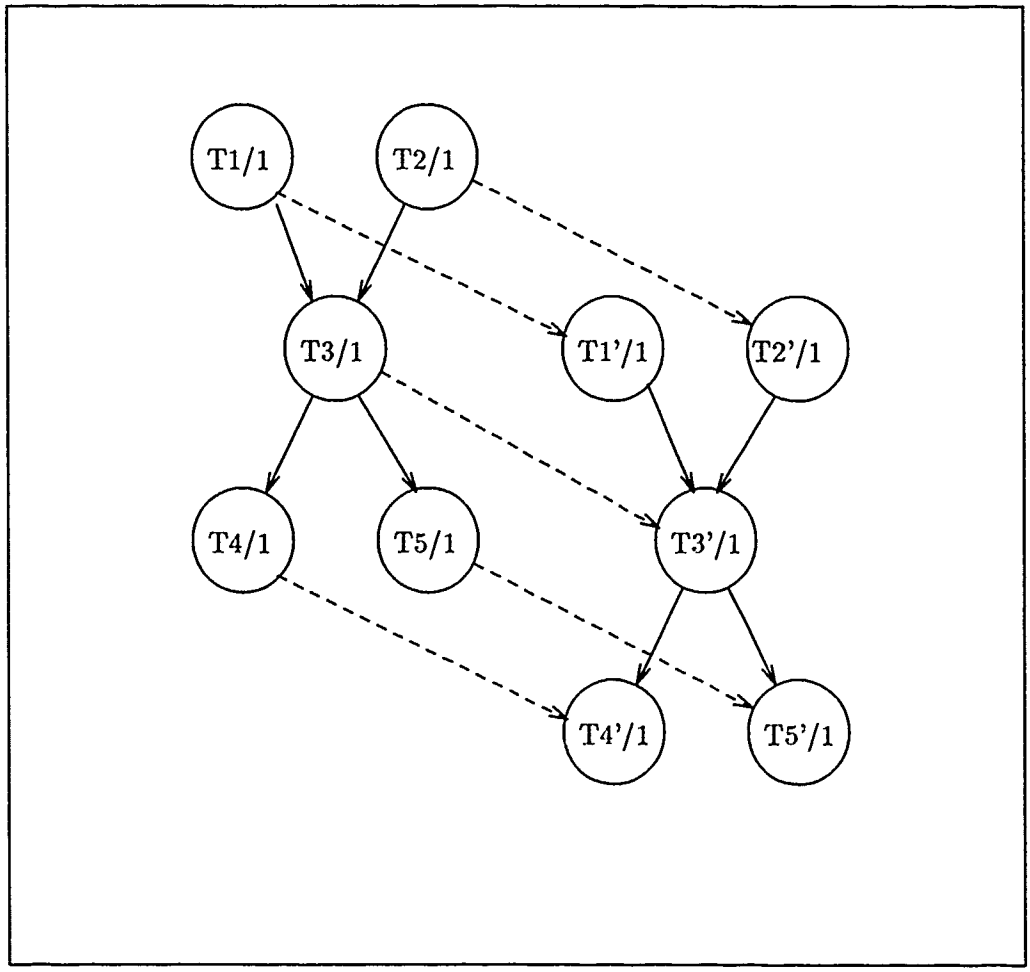



Figure 4.5. Iterative task system

*4.2.3 Modified Backtracking Algorithm* Implementation of the modified backtracking algorithm requires simple data structure management of a stack using a linked list structure since the stack size is predicated on the task system characteristics. This stack is commonly refered to as the OPEN list [31]. The algorithm $A1$ for conducting the modified backtracking search is as follows:

1. Calculate lower bound.

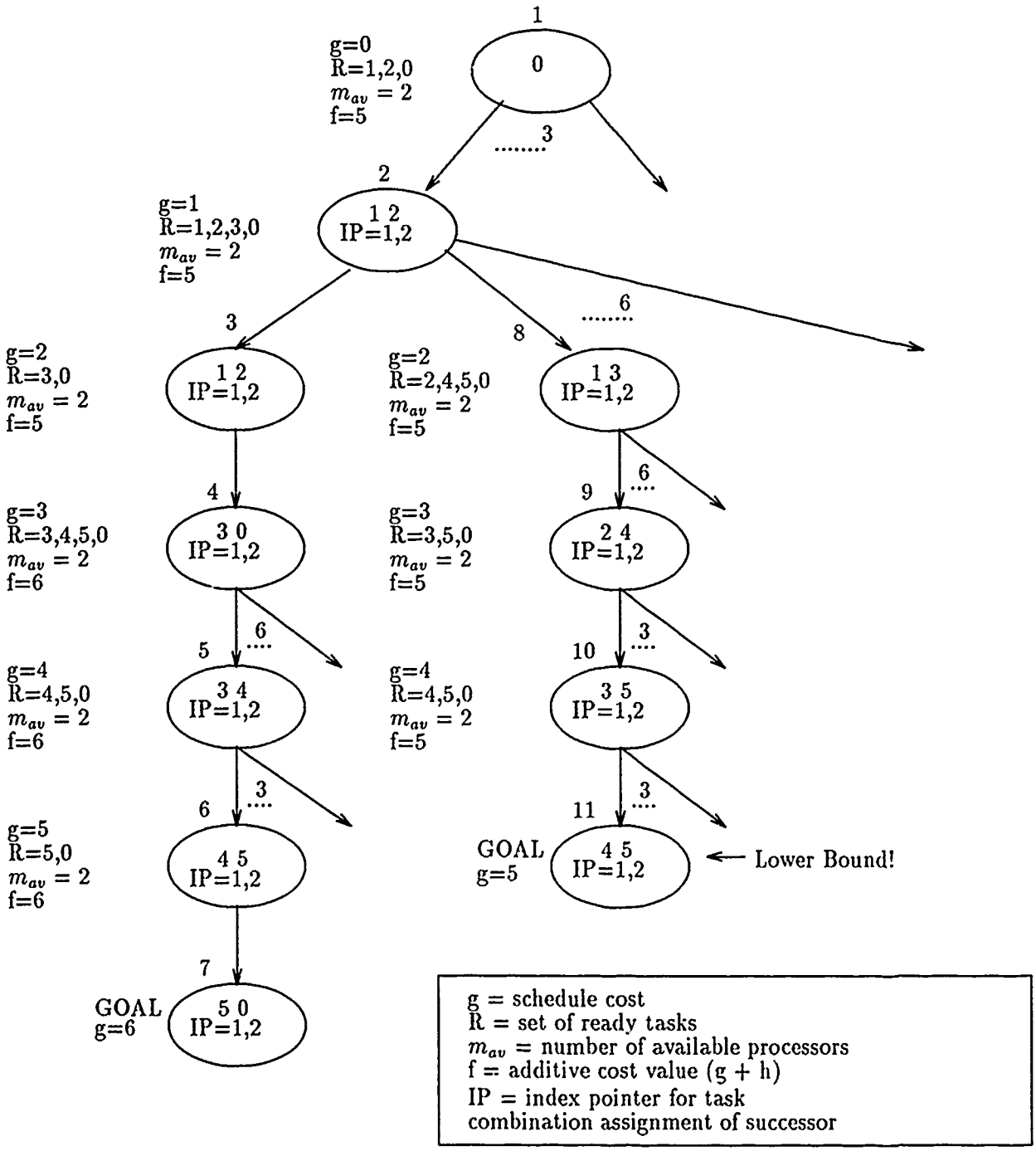



Figure 4.6. Explicit enumeration search tree (control structure) for MBT search.

2. Set upper bound = $\infty$.
3. Generate initial search state and put it on OPEN.
4. Examine topmost node from OPEN.
5. If OPEN is empty, exit.
6. If the heuristic evaluation function $f(n) \geq$ upper bound remove search node from open and go to step 4; otherwise continue.
7. Generate a new successor of the search node and update the combination sequence of ready tasks for future successor generations.
8. If no more generations are possible, remove search node from OPEN.
9. If sucessor is a goal, update upper bound and maintain solution (if necessary); otherwise, discard it.
10. If solution cost of goal = lower bound, exit.
11. If not a solution, calculate branching and heuristic data of successor.
12. If not a solution and $f(n) <$ upper bound, place on top of OPEN; otherwise, discard it.
13. Go to step 4.

*4.2.4 Backtracking Search Implementation* As shown in Figure 4.5, an iterative task system can easily be depicted using a graphical representation of a system where all tasks execute only once (multiple executions of the same task are treated as different tasks). This unfolding of an iterative system allows critical path data to be calculated from each and every task regardless of the system structure. The Floyd-Warshall algorithm with time and space bound $\Theta(n^3)$ is used to calculate the critical path. Although there exists $O(n^2)$ algorithms for calculating longest path values from a single source such as Dijkstra's algorithm [10:527], and the Bellman-Ford algorithm [10:532], the critical path from every node must be calculated increasing the complexity to $\Theta(n^3)$. The code for the Floyd-Warshall algorithm is tight with no elaborate data structures, and so the constant hidden in the $\theta$-notation is small. This critical path data can then be used to help find the best lower bound from each intermediate schedule. However, the one drawback to unfolding an iterative system into unique tasks for these calculations is the large matrix data structure representing path costs between adjacent tasks which results. The bounding expression above becomes $\Theta(n * i)^3)$ where $i$ is the number of iterations required. For example, a task system with 20 tasks and 10 iterations requires an unfolded cost matrix of $200X200$ in size for the above algorithms. Should this space requirement become prohibitive, a depth first recursive procedure may be necessary!

In Figure 4.6, search state information is maintained within each search node in order to generate successor search nodes when necessary. When each successor search node is initially created and found not to be a solution, a list of ready tasks and ready processors is calculated for the next schedule point where at least one task and one processor are ready. This successor is created by assigning a combination of ready tasks to ready processors. The actual tasks which are assigned when a successor search node is generated is determined by the set of combination index pointers, IP. This set is then updated to identify the next combination of tasks to assign if an when it becomes necessary to generate another successor. For example, given three ready processors and two ready tasks, the set of combination IPs is $\{(12),(13),(23)\}$. Search node 8 represents the assignment of ready tasks in the set R of search node 2 when the combination IPs of search node 2 have been updated to the pair (13). The real cost, $g$, of each intermediate schedule is determined from the processor with the longest schedule; *i.e.*, the maximum finish or flow time as described in Chapter 1.

$$g(n) \quad = \quad \max(\omega(p_i))_{i=1..p} \tag{4.3}$$

where $p$ is the number of processors to schedule, $p_i$ is the processor, and $\omega$ is the flow time. The heuristic value, $h(n)$, is calculated as the new lower bound from the current intermediate schedule. Finding a lower bound as close to the upper bound is essential in limiting the search space. Therefore, both bin packing and critical path information are used to aid this process [8] [20] [26].

**Theorem 4.2.3** *Given an iterative task system of arbitrary precedence and variable execution time tasks, the minimum schedule length can be no less than* $\max(\lceil \frac{\sum_{k=1}^{n} \tau_i}{m} \rceil)$, *Critical Path, Current Schedule Length) where n is the number of tasks (including iterations) awaiting scheduling, m is the number of processors being scheduled, and* $\tau_i$ *is the execution time requirement for task* $T_i$ *[38].*

Applying this theorem to the scheduling process provides lower bound values for all non-solution schedules in the search tree. For example, at an intermediate schedule, all the remaining task execution units are summed. Those processors whose schedule lengths are smaller than the schedule flow time are made to be equal by subtracting the difference from this total. The remaining value is then divided by the number of processors with the result rounded up to the nearest integer. This value becomes $h_1$. Also, critical path data from the last task assigned on each processor is applied to that processor's flow time to determine a maximum flow time for the schedule. The largest difference between each processors's new flow time estimate and $g(n)$ for the actual

intermediate schedule becomes $h_2$. The larger of these two values then becomes $h(n)$.

$$h_1 = \lceil \frac{\sum_{i=1}^{n} \tau_i - \sum_{i=1}^{p} (\omega(p)_{max} - \omega(p_i))}{m} \rceil \quad (4.4)$$

$$h_2 = \max(\omega(p_i) + cp(p_i))_{i=1..p} - g(n) \quad (4.5)$$

and

$$h(n) = \max(h_1, h_2) \quad (4.6)$$

where $h(n) \geq 0$, $n$ is the number of scheduled tasks, $\tau_i$ is the execution costs of each unscheduled task, and $cp$ is the critical path. The actual flow time and the heuristic value are then added to produce a lower bound for the intermediate schedule (the additive evaluation function).

$$f(n) = g(n) + h(n) \quad (4.7)$$

An example of these calculations for the task system in Figure 4.7 with the intermediate schedule shown results in

$$h_1 = \lceil \frac{6-2}{2} \rceil$$

$$h_2 = \max(4+2, 2+2) - 4$$

such that

$$f(n) = 6$$

### *4.3 Scheduling State Combinatorics*

Any simulation system which can be developed into unique control (task) entities can be represented using a task graph. The search process must implicitly generate all combinations of search states for all ready tasks at each scheduling point to ensure an optimal schedule is found. The sensitivity of the relationship between the number of ready tasks and the number of ready processors in generating search nodes is fully appreciated when a combination curve is examined.

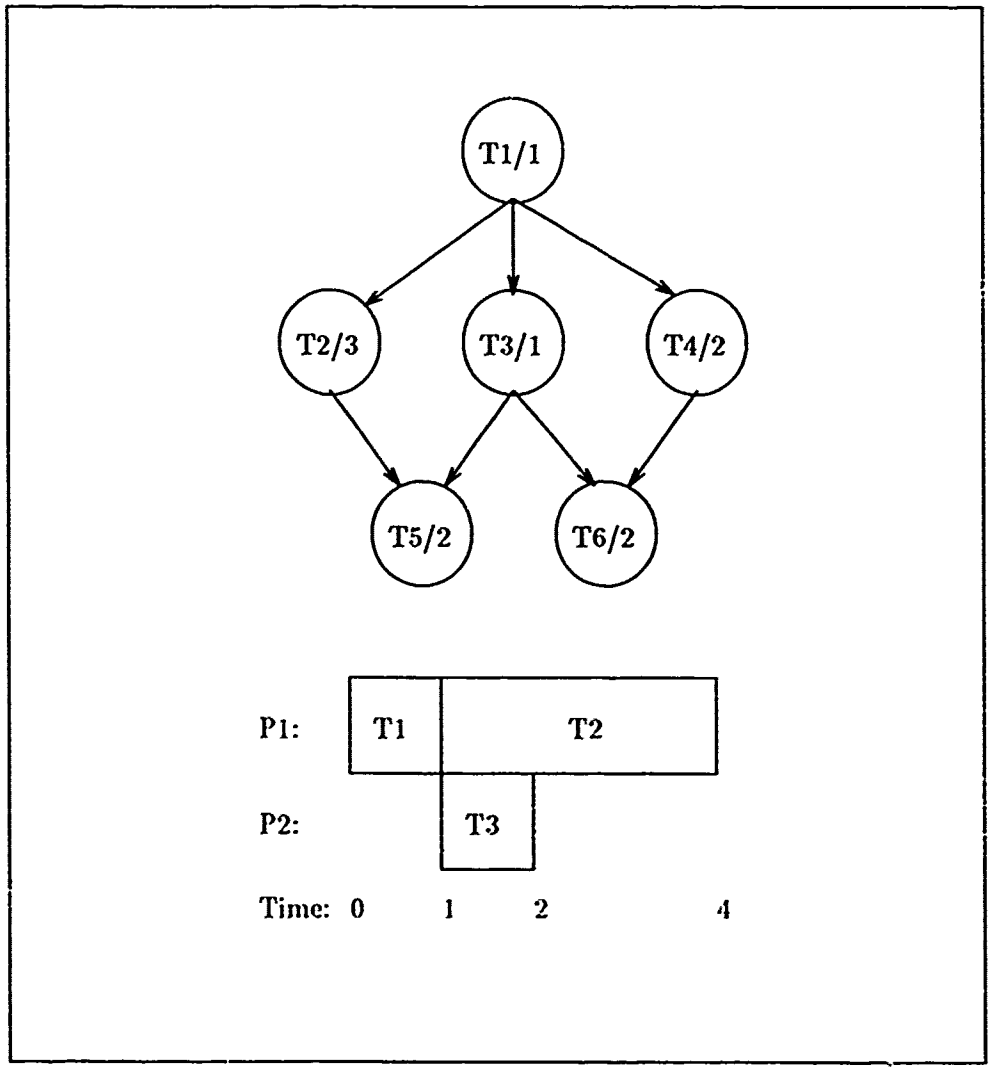



Figure 4.7. Intermediate schedule for lower bound calculations

For example, the largest number of combinations which can be taken from a given number occurs when half the number is taken at a time. Figure 4.8 shows the bell shaped curve which represents this characteristic of numerical combinations. When 20 items are taken 10 at a time, 184,756 combinations exist. Therefore. if 20 tasks are ready to be scheduled, increasing the number of available processors from 1 up to 10 dramatically increases the number of search nodes the search process must implicitly generate, while increasing the number of available processors from 10 up to 20 dramatically reduces this number. In fact, the combination gradient is greatest between 7 & 8 (12 & 13) where number of combinations increases (decreases) by 48,450 for this example. This

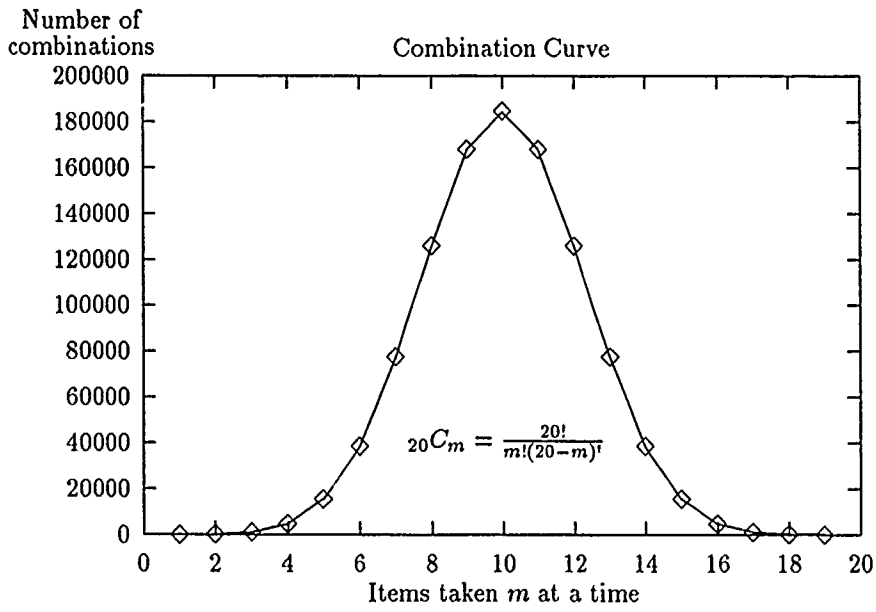



Figure 4.8. Combination curve for 20

dramatic change in the implicit generation of search space based on the availability of one more or one less processor at a scheduling point can greatly affect the search time.

### *4.4 Characteristics of Simulation Systems*

*4.4.1 Sensitivity Analysis* Three important characteristics of simulation systems are (1) forward precedence constraints, (2) feedback precedence constraints, and (3) the difference in task execution times. Modifying a simulation graph by removing or increasing the number of task dependencies, adding feedback dependencies, or modeling a system with equal execution time tasks can all have a significant effect on the size of the explored search space. Figure 4.9 shows a task system with seven EET tasks which must each execute twice when scheduled onto two processor.

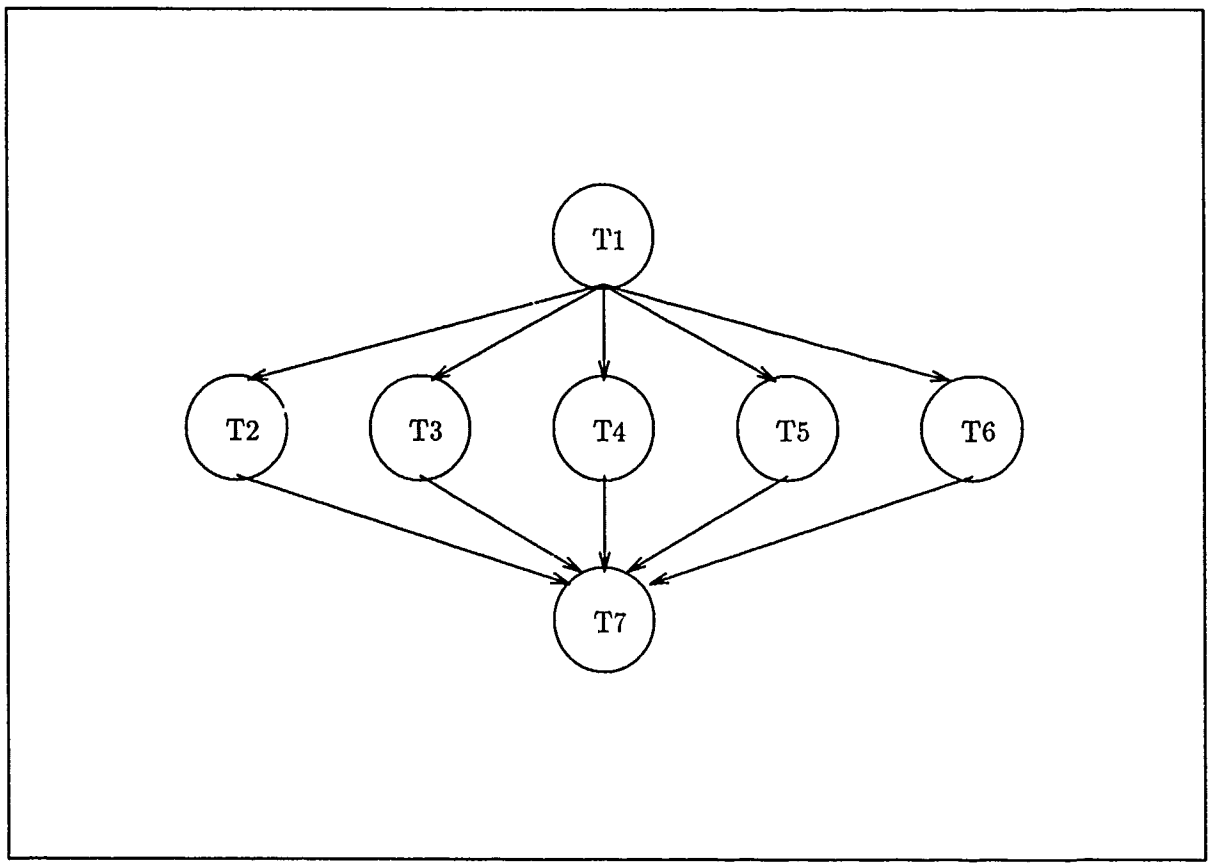



Figure 4.9. Simulation task system.

As Table 4.1 reveals, the size of the enumerated search space varies significantly when these parameters are modified by doubling the execution times for the even numbered tasks, removing a constraint arc from $T1$ to $T4$, or adding a feedback constraint arc from $T7$ to $T1$. All search trees generated assume a modified backtracking process with list scheduling compensation.

Table 4.1. Sensitivity analysis of a simulation system's execution constraints

| Search Nodes Generated | Lower Bound | Optimal Schedule Cost | Feedback $\prec +\{(T7,T1)\}$ | Equal Execution Times (EET) | Relaxed Constraint $\prec -\{(T1,T4)\}$ |
|---|---|---|---|---|---|
| 8 | 7 | 8 | | X | |
| 21370 | 7 | 8 | | X | X |
| 7988 | 7 | 10 | X | X | |
| 17 | 10 | 11 | | | |
| 50590 | 10 | 11 | | | X |
| 502 | 10 | 12 | X | | |

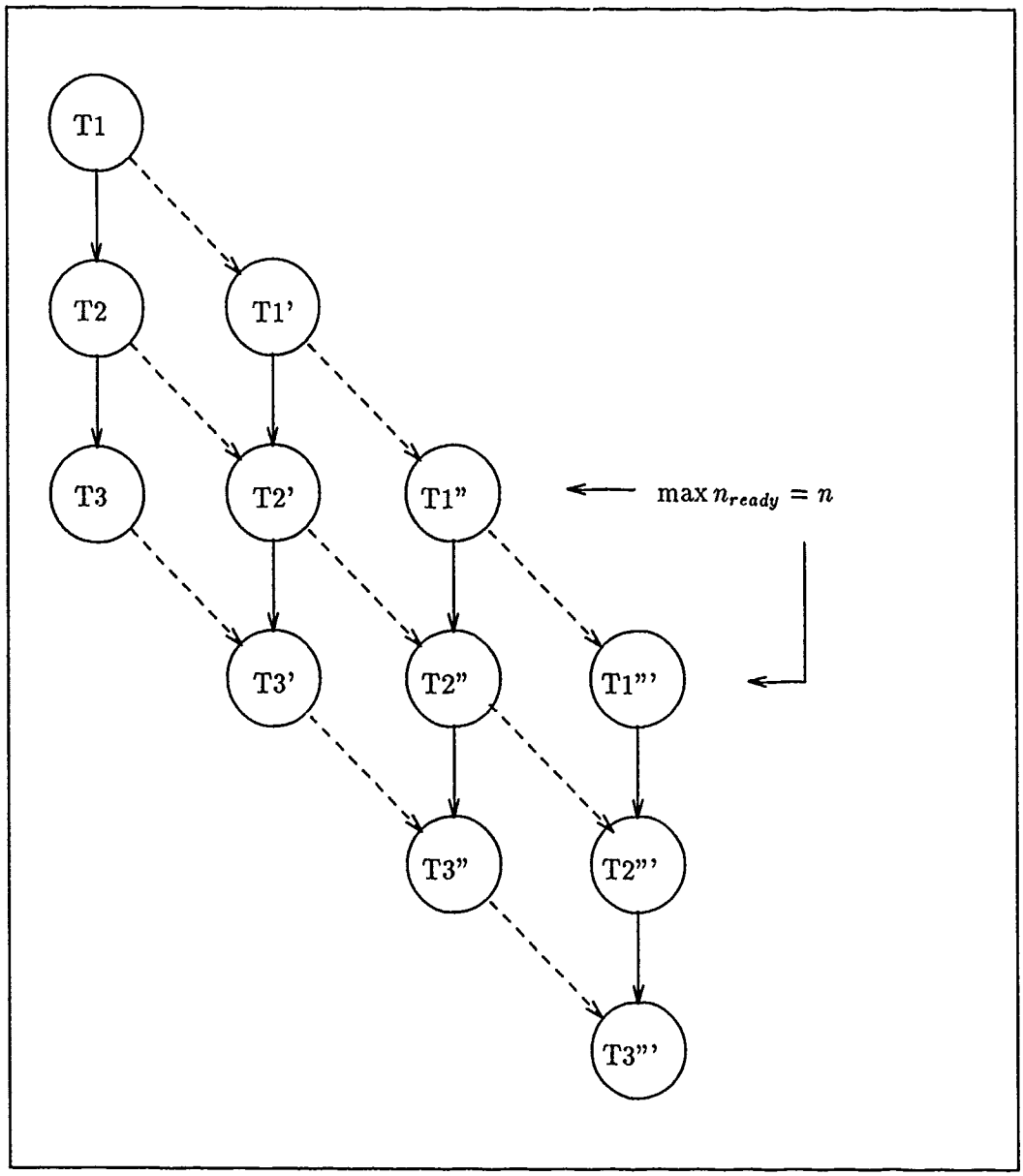
*4.4.2 Minimization of Scheduling Combinatorics* In an iterative EET task system regardless of feedback contraints, the maximum number of tasks available for execution at any one time is the number of tasks in the original system (iterations not included). As shown in Figure 4.10, given enough iterations, each level of the task system can eventually be combined with every other level for scheduling.

**Theorem 4.4.1** *In EET task systems, for* $i = 1 \longrightarrow \infty, \max n_{ready} = n$.

For an EET task system containing $n$ tasks, scheduling at least $n$ processors reduces the number of combinations for each assignment to 1. Therefore, if there are enough processors at each scheduling point to schedule all ready tasks, the task system can be optimally schedule within $\gamma$ discrete step where $\gamma$ is the maximum depth of the search tree; *i.e.,* the number of tasks in the initial critical path including iterations.

**Theorem 4.4.2** *In EET task systems, for* $m \geq n$, *the optimal solution can be found in* $\gamma$ *steps where* $\gamma$ *is the number of tasks on the critical path of the initial task graph including iterations.*

For Figure 4.10, an optimal schedule can be found in six scheduling steps. However, the breadth of the schedule may be such that many tasks which communicate don't physically reside on the same processor. If interprocessor communication costs are considered, such an optimal mapping may not prove to be optimal upon execution of the simulation.




Figur 4. 9. Iterative EET task scheduling example for max $n_{ready} = n$.

### *4.5 Scheduling Process*

The process of making a scheduling decision at each level in the search tree requires knowledge of the free tasks (tasks whose constraints have been relaxed due to completion of parent tasks), and the free processors. The scheduling point must be determined by scanning the free times of the processors in a non-decreasing order until based on the scheduling relationship of the tasks scheduled, at least one task is free. In simulation terminology, this equates to the next-event time. As indicated in Figure 4.11, the next valid scheduling point is at time 3 since task 2 constrains the second interation of itself and the first iteration of task 3 until it completes.

The following algorithm *A2* defines this process:

1. Sort list of processor_ready_times.
2. Scan list of processor_ready_times until ready_time $t$, where $t$ represents the earliest instance at least one task is now unconstrained.
3. List assign first combination of ready tasks to ready processors.
4. Update combination index pointer (IP) for next assignment.

### *4.6 Sequential Search*

The underlying search techniques for generating optimal schedules for simulation task systems vary depending upon whether feedback among the tasks exits, task execution times are identical, task iterations can be scheduled all at once, and task migration among the parallel processors is permitted. As Table 4.2 indicates, seven methods can be imployed in generating an optimal schedule based on this criteria. Each method takes advantage of the characteristics of the task system to produce the smallest search tree possible, and thus, produce an optimal schedule in an efficient manner.

Table 4.2. Sequential search methods

| Method | Feedback | Execution Times | Iterations | Task Migration |
|---|---|---|---|---|
| 1 | No | Equal | Together | No |
| 2 | No | Variable | Separate | No |
| 3 | No | Variable | Separate | Yes |
| 4 | Yes | Equal | Separate | No |
| 5 | Yes | Equal | Separate | Yes |
| 6 | Yes | Variable | Separate | No |
| 7 | Yes | Variable | Separate | Yes |

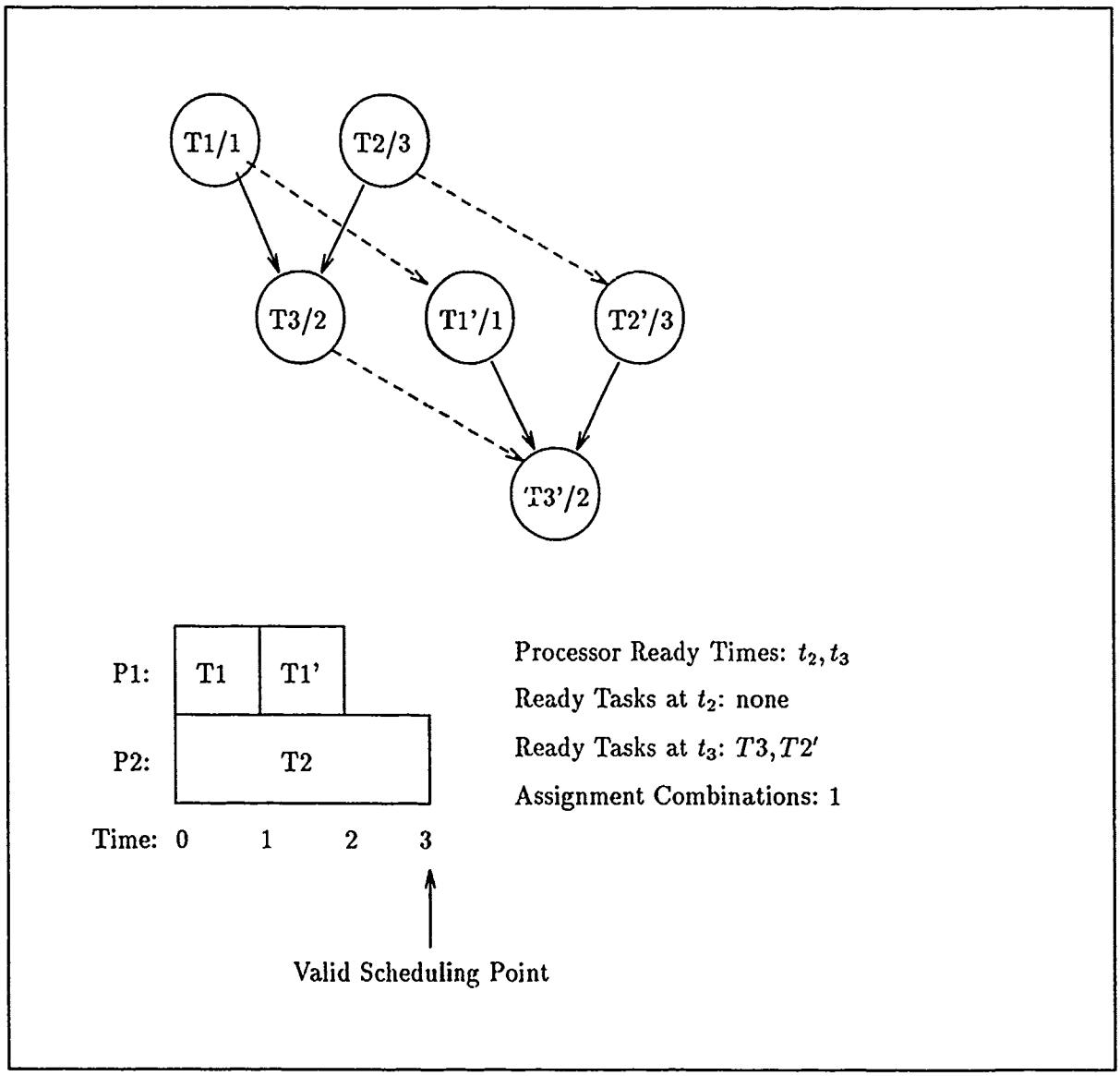



Figure 4.11. Scheduling point decision.

Search method 1 reduces the number of scheduling points and the scheduling combinations which must be implicitly generated. Since systems being modeled with EET tasks don't require compensation for the list scheduling anomalies, this *all-iterations-first* strategy reduces the original complexity of the problem making the relative time costs for obtaining a solution minimal. As shown in Figure 4.12, a three task system can quickly be scheduled onto two processors in four steps (scheduling points) using the *all-iterations-first* strategy in comparison to six steps when each task iteration is treated as a unique task for scheduling purposes.

Search methods 2 and 3 compare the effects of allowing task migration on the scheduling process for variable execution time tasks. Preventing task migration reduces the complexity of exercising the simulation, but it can cause the search tree to be larger in depth and produce schedules whose flow time is greater than schedules with task migration. Interprocessor communication cost at run-time aren't considered in order to speed the schedule generation process and reduce the search space to a size determined by combinations of task assignments at each scheduling point versus permutations.

Search methods 4 and 5 compare the effects of allowing task migration on the scheduling process for equal execution time tasks when feedback constraints exist. Feedback constraints generally increase schedule flow times by introducing processor idle times caused by a feedback focal point. Also, feedback precludes the use of the *all-iterations-first* strategy even though the simulation system contains only EET tasks.

Search methods 6 and 7 compare the effects of allowing task migration on the scheduling process for variable execution time tasks when feedback constraints exist. The performance results are conjectured to be identical to method 2 & 3 above.

### *4.7 Parallel Search*

Optimal schedules for implementation of simulation systems on a parallel, message passing machine such as the *iPSC/2* require implicit search within a potentially large space. The modified backtracking algorithm attempts to reduce this space as much as possible; however, real improvement on search time can only be acheived if the search tree is parsed into smaller sections where simultaneous searching of these sections can occur. Such is the intent of parallelizing the sequential search process.

Three parallel techniques are explored to identify the most efficient method. As Table 4.3 indicates, each method differs from the other based on upper bound communications and load balancing activities. All three methods use a distributed-list approach for machine scalability.

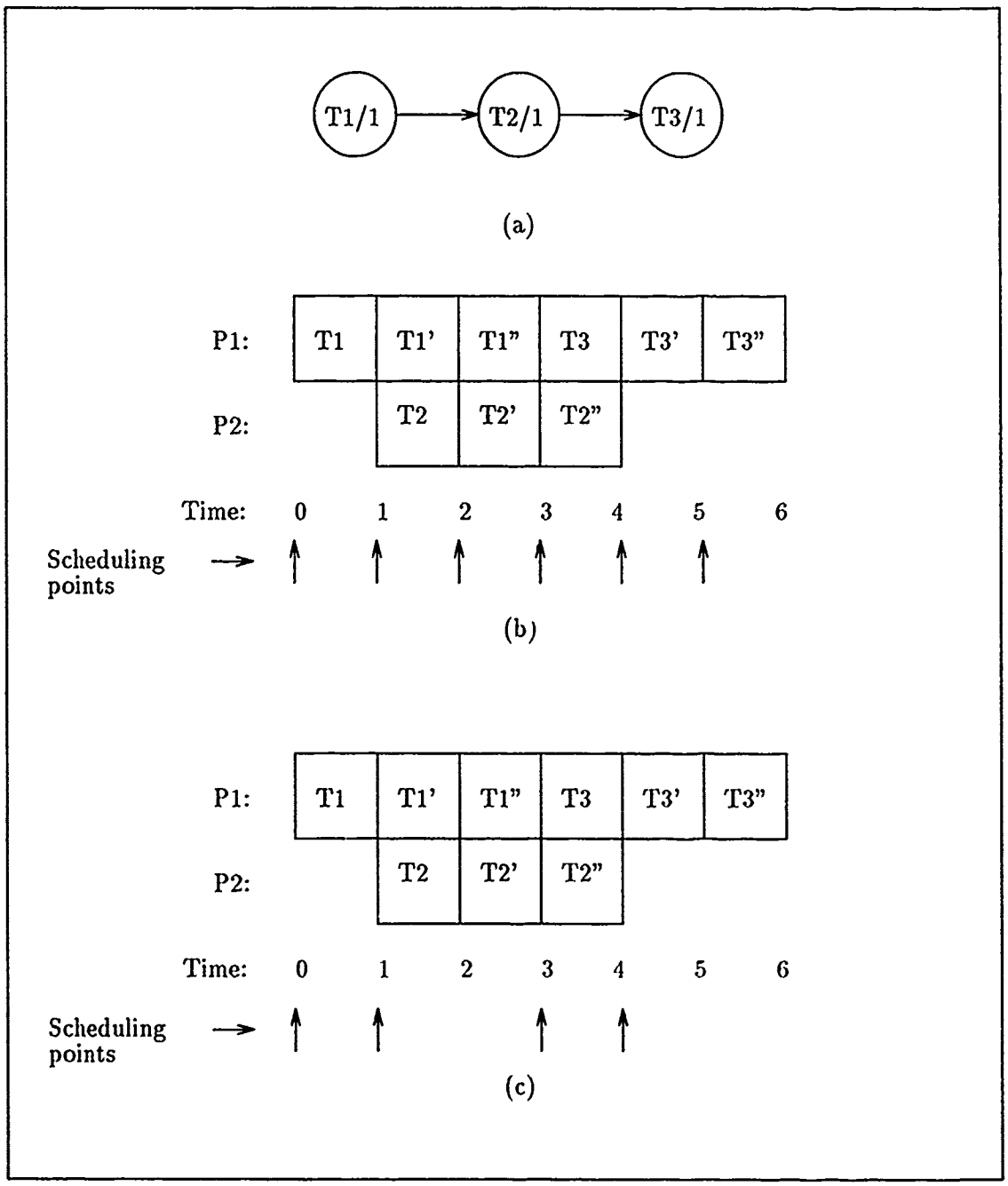



Figure 4.12. Scheduling simplification for EET task systems without feedback. (a) Task system requiring three iterations. (b) Each iteration assigned separately. (c) All iterations assigned together (*all-iterations-first* strategy).

Such an approach doesn't suffer from the bottleneck effect of the centralized list approach when many processors are assigned to the search process [12].

Table 4.3. Parallel search methods

| Method | Data Decomposition | Upper Bound Communications | Load Balancing |
|---|---|---|---|
| 1 | distributed list | | |
| 2 | distributed list | X | |
| 3 | distributed list | X | X |

In all parallel search methods, each physical processor starts by generating search nodes in a modified breadth-first manner; *i.e.*, only the $p$ most search nodes at the top of the search tree are generated where $p$ is the number of physical processors being used to find an optimal schedule. At this point, each process keeps the search node whose position on the OPEN list corresponds to its node number and discards the rest. For instance, if there are eight processors available, the initial modified breadth-first search conducted on each processor will produce an OPEN list containing eight search nodes on each processor P. P1 will keep the $1^{st}$ search node, P2 will keep the $2^{nd}$, P3 the $3^{rd}$, etc ... The modified backtracking search is then initiated and proceeds until an optimal solution is found.

In parallel search method 1, once each processor has an initial search node to work from, all physical processors proceed in their search unabated until all processors run out of search space to explore. This is conjectured to produce an inefficient search where additional search space is unneccessarily explored and machine utilization is poor due to load imbalancing. In parallel search method 2, local upper bound information is passed on to the other working processors to aid in the global pruning process of the local search trees. This is conjectured to reduce the explored search space but with little or no improvement in machine utilization efficiency. In parallel search method 3, load imbalancing is corrected with *on-demand* movement of search nodes to idle processors. This method is conjectured to greatly improving machine utilization efficiency an produce an optimal schedule in the minimum amount of time. All three parallel search methods require an exhaustive search, but with upper bound communications and load balancing activities, the global search tree enumerated should be a fraction of the search tree enumerated when such activities don't occur.

Since the search process is exhaustive, all working processors must implicitly explore their associated initial search space under methods 1 & 2 before an optimal solution can be assertained. However, in method 3, load balancing causes each processor's initial search space to change when load balancing requests are satisfied. Such requests are satisfied by sending the search node highest

in the search tree to the requesting processor. This attempts to amortize the cost of satisfying the load balancing request by ensuring as much potential search space as possible is provided to the requesting processor. A counter is then initialized so that a specified number of search expansions occur before another load balancing request is satisfied by the same processor. This number is determined to be the worst case depth of the local search tree $n * i$, where $n$ is the number of tasks, and $i$ is the number of iterations regardless of the actual task system structure. This worst case value ensures that the processor has an opportunity to find at least one solution prior to again losing data to a load balancing request. Such a threshold reduces load balance thrashing between the processors and provides the global search process the potential for a new upper bound value within the threshold limit. For example, Figure 4.13 shows a simple simulation system with three EET tasks. Each task must execute for three iterations. When scheduling this system onto three processors, nine scheduling decisions must be made to find a complete schedule when the feedback constraint is assumed (Figure 4.13, part (b)), whereas only five scheduling decisions must be made when feedback isn't assumed (Figure 4.13, part (c)). Each scheduling decision represents a new level in the search tree for this example. Thus, if the structure of the actual task system results in a maximum search tree depth of five, at least one new solution value is guaranteed the opportunity to be discovered. Due to heuristic pruning though, actually realizing a new solution may not occur. The actual maximum depth of the search tree isn't considered since the primary objective of the threshold value is to delay satisfying load balancing requests until at least one new solution is given the opportunity to be found.

During the load balancing process, when a processor identifies an empty local OPEN list, it makes a request to its nearest logical neighbor. Should the requesting processor receive a negative reply, it then makes the same request to the logical neighbor of the processor which denied the request. This request proceeds in a spoke fashion as shown in Figure 4.14 until either it is satisfied by another processor, or all processors have been polled. Such a strategy is simple to implement and doesn't suffer from excessive communication costs. The summation of links traversed in communications with all processors in a logical order is no different than when communications proceed in a physical order based on increasing path lengths expanding out from the starting processor. Figure 4.15 shows the processor interconnection topology for a hypercube with eight processors. As Table 4.4 shows, the total number of links traversed for both methods is the same. Although the intermediate sums may favor the later communications order as indicated by the link totals through the third load balancing request, the communication latency for an $iPSC/2$ varies by at most 10% between any two processors [13.441], *i.e.*, the added communications latency is relatively insignificant considering the simplicity of the implementation.

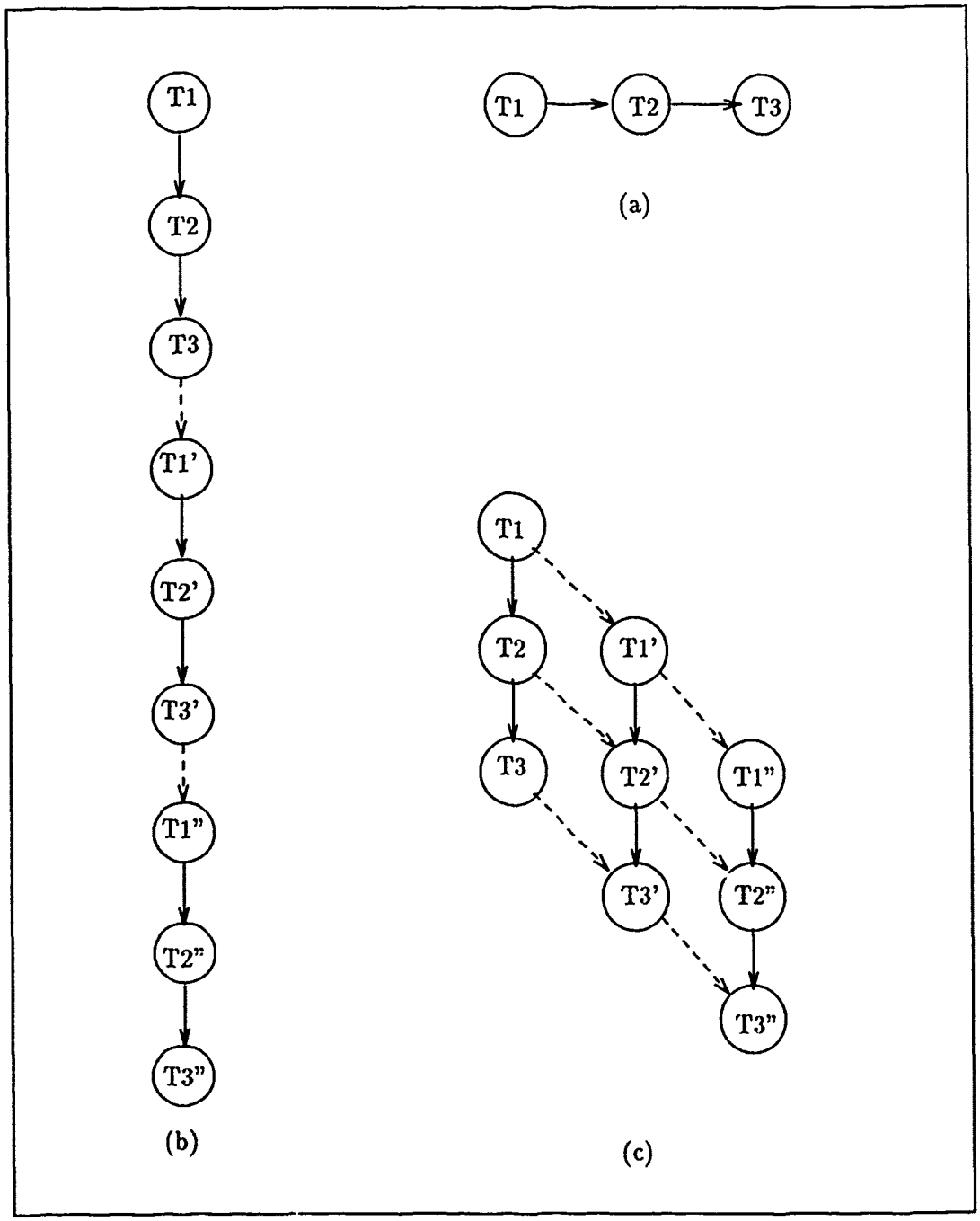



Figure 4.13. (a) Simulation task system. (b) Iterative graph for $\prec + \{(T3, T1)\}$. (c) Iterative graph without feedback.

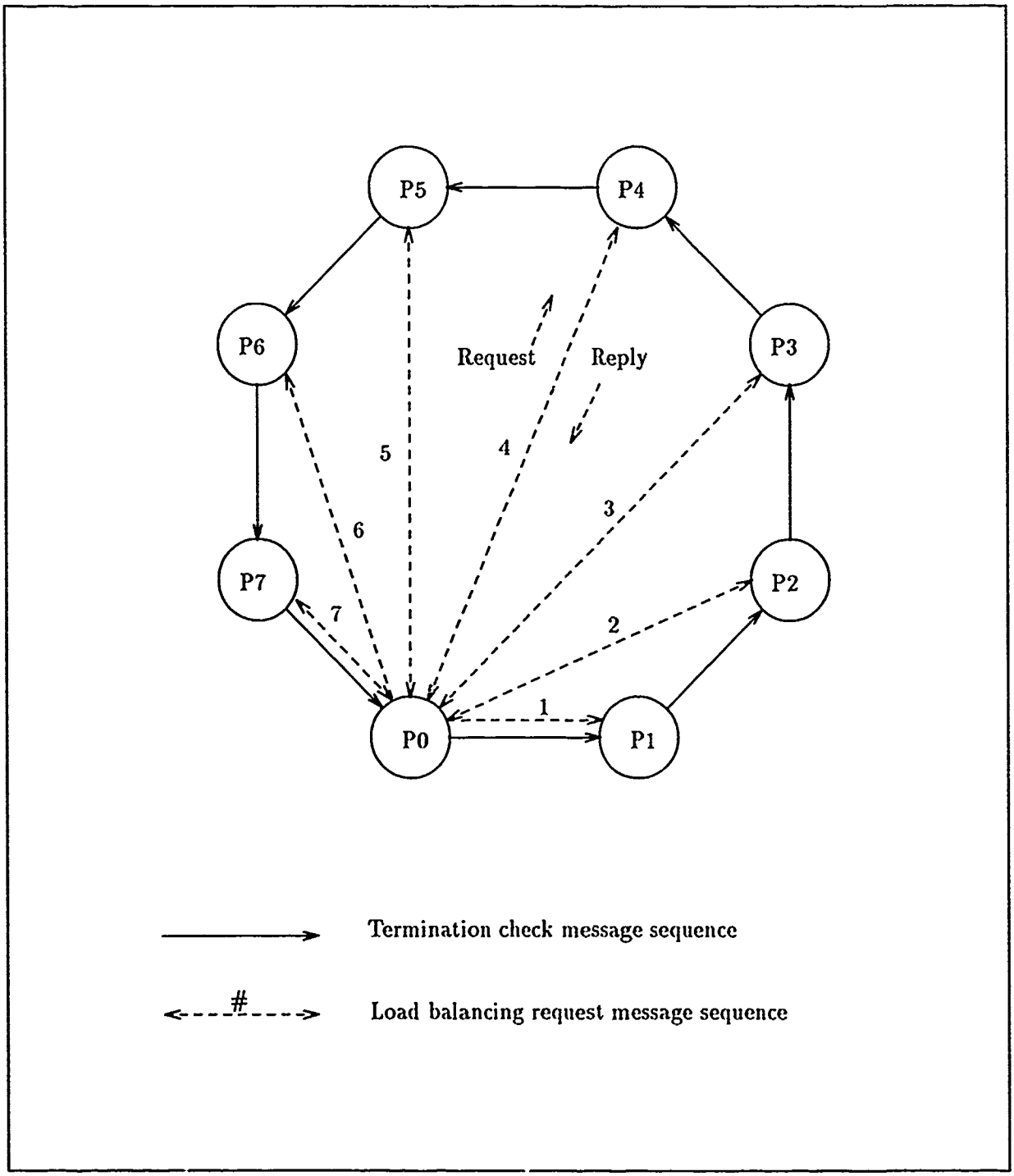



Figure 4.14. Parallel load balancing and termination communication structure.

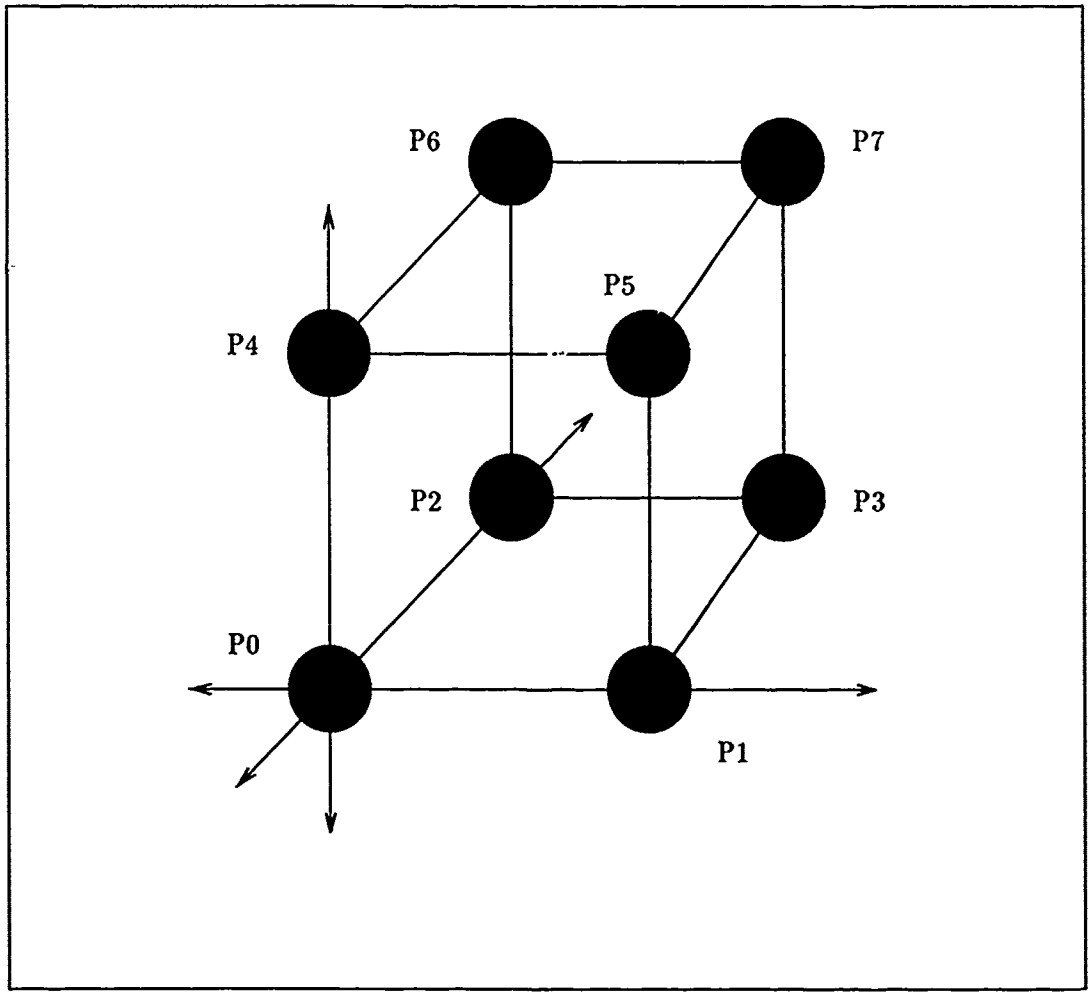



Figure 4.15. Interconnection topology for a hypercube with 8 processors.

Table 4.4. Communication path lengths for the hypercube interconnection topology starting from processor 0.

| Logical Order | | | Increasing Path Length Order | | |
|---|---|---|---|---|---|
| Destination Processor | Links | Accumulative Link Totals | Destination Processor | Links | Accumulative Link Totals |
| 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 | 1 | 2 |
| 3 | 2 | 4 | 4 | 1 | 3 |
| 4 | 1 | 5 | 3 | 2 | 5 |
| 5 | 2 | 7 | 5 | 2 | 7 |
| 6 | 2 | 9 | 6 | 2 | 9 |
| 7 | 3 | 12 | 7 | 3 | 12 |

If the case arises wherein all processors have been polled and all have denied the load balancing request, a termination check message is initiated in the same ring order. Each processor will receive the termination check message and pass it on to their logical neighbor unless they are still working; *i.e.*, their OPEN list isn't empty. Each termination check message is uniquely identified by the processor ID which initiated the message. If a processor receives a termination check message with the same ID as itself, the message has successfully traversed the ring network indicating all other processors are idle too, at which point it broadcasts a terminate message so the processor network can synchronize, collect performance data, and identify the optimal solution. Figure 4.14 represents this termination sequence initiated by processor $P0$ for an 8 processor configuration.

The algorithms for the three parallel versions identified in Table 4.3 are all slight modifications of the MBT algorithm to account for load distribution, load balancing, and communications requirements. Each algorithm executes the same initialization as follows:

1. Calculate lower bound.
2. Set upper bound = $\infty$.
3. Generate P search states and keep $P_i$ search state on OPEN.

When each algorithm exits on all processors, global data collection routines are invoked to obtain the global optimal solution from among all the local optimal solutions, along with performance data for analysis of the search process. The following algorithm $PA1$ executed on each processor describes parallel method 1:

1. Examine topmost node from OPEN.
2. If OPEN is empty, execute global minimum to determine global optimum solution from all local optimum solutions, then exit.
3. If the heuristic evaluation function $f(n) \geq$ upper bound remove search node from open and go to step 1; otherwise continue.
4. Generate a new successor of the search node and update the combination sequence of ready tasks for future successor generations.
5. If no more generations are possible, remove search node from OPEN
6. If successor is a goal, update upper bound and maintain solution (if necessary); otherwise, discard it.
7. if solution cost of goal = lower bound, exit.

8. If not a solution, calculate branching and heuristic data of successor.
9. If not a solution and $f(n) <$ upper bound, place on top of OPEN; otherwise, discard it.
10. Go to step 1.

Note that this algorithm is the same as the MBT algorithm described in Section 4.2.3. The following algorithm $PA2$ executed on each processor describes method 2 with the upper bound communications requirement:

1. Examine topmost node from OPEN.
2. If OPEN is empty, execute global minimum to determine global optimum solution from all local optimum solutions, then exit.
3. If the heuristic evaluation function $f(n) \geq$ upper bound remove search node from open and go to step 1; otherwise continue.
4. Generate a new successor of the search node and update the combination sequence of ready tasks for future successor generations.
5. If no more generations are possible, remove search node from OPEN.
6. If successor is a goal, update upper bound, maintain solution, and broadcast new upper bound value to other processors (if necessary); otherwise, discard it.
7. If solution cost of goal = lower bound, exit.
8. If not a solution, calculate branching and heuristic data of successor.
9. If not a solution and $f(n) <$ upper bound, place on top of OPEN; otherwise, discard it.
10. Poll for upper bound message and update local upper bound value when received (if necessary).
11. Go to step 1.

Note the simple additions of broadcasting upper bound data to the other processors and polling for these messages. Since method 3 incorporates load balancing to improve machine utilization efficiency, processors must wait after their OPEN lists become empty until the proper termination sequence has successfully completed. This causes the MBT algorithm to grow considerably in complexity as defined by algorithm $PA3$ below:

1. Examine topmost node from OPEN.

2. If OPEN is empty, send work request to another processor and go to step 10.
3. If the heuristic evaluation function $f(n) \geq$ upper bound remove search node from open and go to step 1; otherwise continue.
4. Generate a new successor of the search node and update the combination sequence of ready tasks for future successor generations.
5. If no more generations are possible, remove search node from OPEN.
6. If successor is a goal, update upper bound, maintain solution, and broadcast new upper bound value to other processors (if necessary); otherwise, discard it.
7. If solution cost of goal = lower bound, exit.
8. If not a solution, calculate branching and heuristic data of successor.
9. If not a solution and $f(n) <$ upper bound, place on top of OPEN; otherwise, discard it.
10. Poll for termination check message and pass it on to next processor if OPEN list is empty. If termination check message originated from this processor, broadcast terminate message and exit.
11. Poll for upper bound message and update local upper bound value when received (if necessary).
12. Poll for work reply message. If positive, place search node on OPEN and go to Step 1. If negative and all processors have responded, initiate termination check message and go to step 10; otherwise, go to step 2.
13. Poll for work request message. Send search node to requester if load balance threshold is exceeded; otherwise, send negative reply message.
14 Poll for terminate message. If received, exit.
15. Go to step 1.

### *4.8 Software Development using Ada*

Developing a software program to exercise and evaluate the algorithmic ideas contained within this research is a complex task. However, using the functional design methodology and coding the modules in Ada which has considerable expressive power reduced the potential effort considerably [5]. This provided greater time for analyzing the search process and performance efficiency of the *iPSC/2* hypercube running the parallel search algorithms.

*4.8.1 Functional Design* Functional design decomposition involves considering a system as a set of interacting functional units. Since the most important design quality attribute is maintainability, maximizing cohesion in a software component and minimizing the coupling between software components is essential in achieving that goal. Although the software developed during this research may be relatively short-lived, these attributes of a good functional design are very valuable in rapidly developing operational code and making the necessary modifications for the various types of scheduling methods requiring investigation. Once all the procedural operations are developed, orchestrating their activities to perform the modified backtracking algorithm as defined in this chapter is a simple matter.

*4.8.2 The Ada Language* Many languages satisfy a collection of requirements:

- Structure Constructs.
- Strong Typing.
- Relative and absolute precision specifications.
- Information hiding and data abstraction.
- Concurrent processing.
- Exception handling.
- Generic definitions.
- Machine-dependent facilities.

However, Ada brings all these elements together in a single language uniting them into one coherent model making program application development comparatively less complex. Structured constructs, strong typing, and information hiding and data abstraction are key elements fully taken advantage of in the development of the parallel search programs. Such attributes allow early detection of many programming errors at compile time versus spending many hours debugging semi-operational code. Code maintenance is also simplified due to the structured constructs allowing code modifications well after initial development to be relatively easy.

Within the development environment of the *iPSC/2*, the Ada constructs for managing the interprocessor communications are very similar to the two other programming languages available for this machine, C and Fortran. However, a firm understanding of the communications hardware is necessary to ensure that the programmer is in control, and not the hypercube!

## 4.9 Summary

This chapter focuses on the MBT search technique and its algorithmic implementation. Since the maximum number of search nodes which must be maintained during the search process at any one time is the maximum depth of the search tree, this method alleviates the problem of machine storage limitations in practical applications. The MBT search enumerates a much smaller search tree than the standard backtracking method with delayed termination resulting in reduced search times. This results from the use of heuristic, lower bound, and upper bound information. The actual number of search nodes created for Figure 4.5 are shown to be reduced by a factor of over 10,000 when the MBT method is compared against the standard BT method with delayed termination. Unfortunately, sensitivity analysis of system data reveals the poorly behaved scheduling combinatorics where minor modifications of the system cause wide variations in the number of search nodes explicitly generated by the MBT method. *A priori* analysis of the simulation task system is also shown to be an important factor in determining how an optimal schedule is built. Task execution times, iteration groupings, and task migration allowances are important characteristics of simulation systems in producing optimal schedules in the most efficient manner. Three parallel techniques are described for implementing the MBT search algorithm on an *iPSC/2* hypercube with the Ada programming language. Methods 2 & 3 build on method 1 and increasingly traded simplicity for machine efficiency to improve the potential speed-up of the search process. The use of Ada in developing programs to exercise the parallel MBT algorithms on the hypercube was very beneficial in rapidly developing operational code. The structured design approach fit well with the algorithmic description of the MBT method allowing the operational programs to be well structured and easily maintainable.

# V. Simulation Applications/Search Performance Results

## 5.1 Introduction

Task systems for simulations vary significantly based on what is being simulated. In war gaming simulations, task execution costs vary from one task to another, tasks repeat execution, and feedback constraints exist. The logical partitioning of activities for such simulations creates tasks which simulate actual organizations within a military unit from the headquarters down to the field units which must carry out the orders and provide battle status information (feedback) to the commanders. In VHDL circuits, feedback of circuit information is often used for control of circuit activity. However, many circuits can be implemented on parallel computers without feedback. An example of such a circuit is an 8-bit adder which adds numerous pairs of numbers in a pipelined manner. Although each task executes many times, no information is passed back to any parent tasks. Also, the task execution costs can often be assumed identical due to the simplicity and commonality of the tasks. Therefore, when searching for an optimal schedule for such task systems, such information is useful in reducing the search time as outlined in Chapter 4. When the system is actually implemented on a parallel computer, task migration must also be addressed. If the difference in communication costs between tasks on different processors and tasks on the same processor are negligible, task migrations can be allowed in the scheduling process further minimizing the schedule length. Since the most difficult scheduling problems to solve are those classified as $NP$-complete, the iterative scheduling problem with variable execution time tasks and feedback constraints is examined for search time performance on an $iPSC/2$ hypercube.

## 5.2 Task Labeling

Task labeling is the process of assigning task numbers to the tasks. The task numbering convention chosen is such that arcs between tasks labeled in increasing order represent feedforward dependencies while arcs between tasks labeled in decreasing order represent feedback dependencies. This labeling scheme is necessary to identify which tasks precede others in the adjacency matrix for the schedule generation process when feedback is considered. The feedback information is maintained in the lower triangle of the adjacency matrix with the upper triangle maintaining the feedforward information.

## 5.3 Testing Methodology

Validation of the parallel MBT algorithm requires experimental analysis of the run-time output to ensure the algorithm is free from errors and complies with its requirements in its im-

plementation form. The requirements are to generate optimal schedules for iterative task systems with arbitrary precedence-constraints (feedback included), and variable task execution times using parallel techniques to improve the search time. Since the scheduling combinatorics exhibit great sensitivity to these task system parameters as described in chapter 4, the uniqueness of each conceivable example in its search space and time values prohibits meaningful comparison among them. A true validation of algorithm performance is made by varying the parallel run-time parameters using the same input data. The run-time parameters adjusted for validation of the parallel MBT algorithm are the number of processors used in the search process and interprocessor communications and load balancing activities.

### *5.4 Generic Simulation*

Figure 5.1 represents a computer simulation known as the car wash [42]. This is essentially a queueing problem designed to analyze computer performance based on task partitioning and assignment to a varying number of processors. To validate parallel search methods 1 through

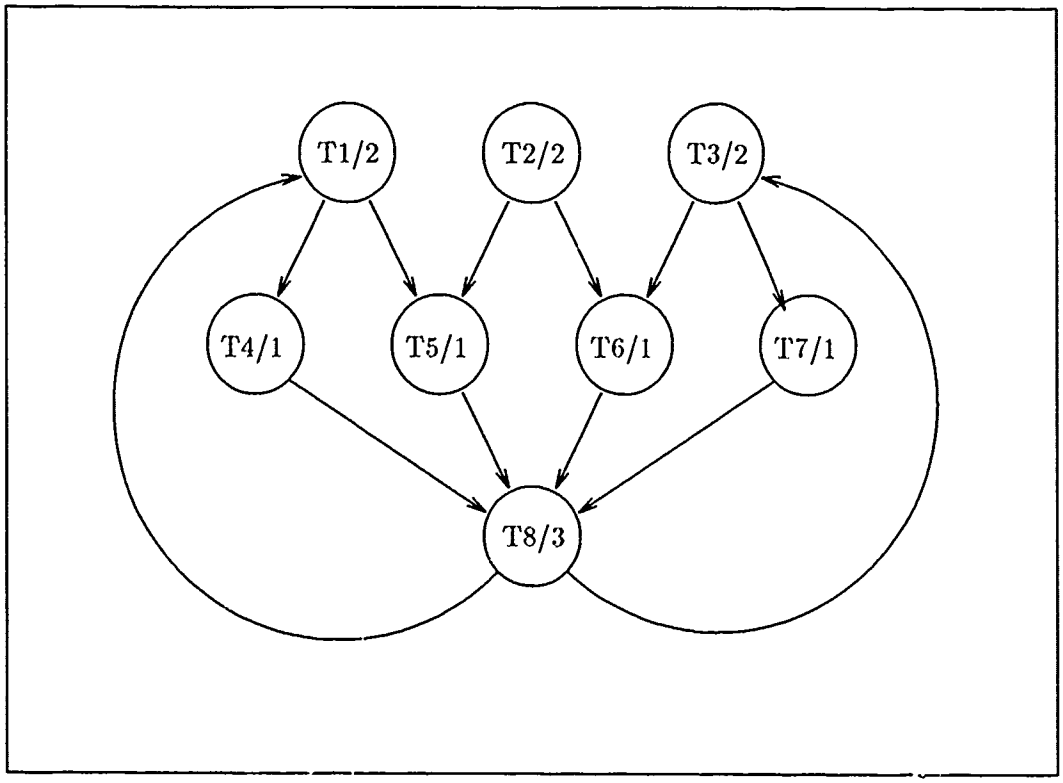



Figure 5.1. Computer simulation of a car wash.

3 for maximum performance, the car wash simulation for three iterations is scheduled onto two processors of the *iPSC/2* hypercube using the same computer. As indicated in Table 5.1, the best search time and load balancing is achieved using parallel search method 3 ($\mu$ represents the average run-time for the processors, and $\sigma$ is the standard deviation of those run-times). When the

Table 5.1. Search time performance using 8 processors to schedule 2.

| Method | $\mu$ (sec) | $\sigma$ (sec) | Search Nodes Created | Upper Bound Messages | Load Balancing Requests | Load Balancing Satisfied |
|---|---|---|---|---|---|---|
| 1 | 167.0 | 85.00 | 71,972 | 0 | 0 | 0 |
| 2 | 44.4 | 15.20 | 18,909 | 6 | 0 | 0 |
| 3 | 44.2 | 0.03 | 18,870 | 6 | 405 | 121 |

number of processors used to create the optimal schedule for two processors is varied, the search performance results further support parallel search method 3 as the 'best' method when considering search space and time.

Table 5.2. Search time performance using 4 processors to schedule 2.

| Method | $\mu$ (sec) | $\sigma$ (sec) | Search Nodes Created | Upper Bound Messages | Load Balancing Requests | Load Balancing Satisfied |
|---|---|---|---|---|---|---|
| 1 | 188.0 | 61.80 | 41,524 | 0 | 0 | 0 |
| 2 | 85.0 | 20.00 | 18,647 | 3 | 0 | 0 |
| 3 | 86.0 | 0.02 | 18,648 | 3 | 83 | 28 |

Table 5.3. Search time performance using 2 processors to schedule 2.

| Method | $\mu$ (sec) | $\sigma$ (sec) | Search Nodes Created | Upper Bound Messages | Load Balancing Requests | Load Balancing Satisfied |
|---|---|---|---|---|---|---|
| 1 | 188.0 | 37.40 | 20,731 | 0 | 0 | 0 |
| 2 | 123.2 | 27.80 | 13,541 | 3 | 0 | 0 |
| 3 | 124.5 | 0.04 | 13,539 | 3 | 28 | 7 |

Parallel machine efficiency is defined as the ratio of parallel speed-up to the number of processors used:

$$\varepsilon = \frac{S_p}{P} \tag{5.1}$$

Table 5.4. Search time performance using 1 processor to schedule 2.

| Method [a] | $\mu$ (sec) | $\sigma$ (sec) | Search Nodes Created | Upper Bound Messages | Load Balancing Requests | Load Balancing Satisfied |
|---|---|---|---|---|---|---|
| 1 | 329.6 | - | 18,147 | 0 | 0 | 0 |

[a]Methods 2 & 3 don't apply

where $S_p$ is the speed-up obtained and $P$ is the number of processors. Table 5.5 shows the $\epsilon$ values for the car wash schedule generation results.

Table 5.5. $\epsilon$ values for car wash schedule generation.

| Processors P | Search Nodes Created | $T_p$ (sec) | $T_s$ (sec) | $S_p$ | $\epsilon$ (%) |
|---|---|---|---|---|---|
| 2 | 13,539 | 124.5 | 329.6 | 2.64 | 1.32 |
| 4 | 18,648 | 86.0 | 329.6 | 3.83 | 0.96 |
| 8 | 18,870 | 44.2 | 329.6 | 7.46 | 0.93 |

As indicated in Table 5.5, maximum parallel efficiency occurs when two processors are used in the search process. In fact, the efficiency is greater than 1 resulting in a speed-up for two processors greater than linear. Also, the parallel efficiency decreases when more processors are added to solve the problem. Since the ultimate goal of using a parallel computer is to reduce the run-time by at least a linear factor in proportion to the number of processors used, these results at first are disturbing. However, due to the nature of the search process, this is expected!

When applying a search process using data decomposition techniques, the overall run-time of the process isn't the only indication of algorithm performance. Another indicator is the enumerated search space actually explored. In this case, using only two processors resulted in the least search space explored to obtain the optimal solution. In fact, the increasing order of $\epsilon$ is directly related to the number of search nodes created as Tables 5.1-5.3 indicate. In general, using more processors reduces parallel machine efficiency slightly due to the potential for duplication of search efforts caused by lack of global knowledge by the processors at each step in the search node generation process. However, significant speed-up of the search process can still be obtained.

Another performance metric for the parallel MBT algorithm is the number of search nodes generated per processor for a given problem per unit time.

$$\beta = \frac{search\_nodes\_created}{P * \mu} \tag{5.2}$$

where $\beta$ = the number of nodes generated per processor per second. Given the data in Tables 5.1-5.3 for parallel method 3, $\beta = 54.25$ with $\sigma = 0.6$ for the four samples used.

### *5.5 VHDL Simulations*

Using VHDL on a parallel machine to simulate VLSI circuits is very important in reducing the time spent on circuit design and analysis. Scheduling VHDL tasks onto a parallel computer in an optimal sequence ensures a minimum simulation run-time for a given analysis requirement. VHDL circuits designed for implementation on an *iPSC/2* hypercube [9] are used as source data for the 'best' parallel search algorithm, parallel method 3. One circuit is a carry-lookahead adder which has 30 VET tasks each requiring 4 iterations. A search time limit of 600 seconds was set to evaluate the search performance. Unfortunately, an optimal solution couldn't be found within this time limit. Since general simulation problems are $NP$-hard with exponential search time requirements, such limitations are necessary to prevent excessive delays in aquiring data while providing the 'best' solution within the desired time limit. As indicated in Table 5.6, the search limit of 600 seconds is reached when attempting to schedule the simulation onto 8 processors using an $n$ dimension hypercube where $n$ varies from 0 to 3. Although the schedule length of 68 is only within 21% of the lower bound for all runs, no guarantee is made that the lower bound can be achieved given enough time. In fact, the solution reported may be the optimal solution given the precedence-constraint relationship of the simulation system. Unfortunately, this can only be guaranteed if the search process is given the time to explore all viable scheduling possibilities; *i.e.*, the process terminates on its own. When the input data is changed to schedule 4 processors instead of 8, the 600 second

Table 5.6. Search performance to schedule a carry-lookahead adder onto 8 processors.

| Processors Used | Search Time (sec) | Lower Bound | Schedule Length | Search Nodes Generated | $\beta$ | $\mu_\beta$ | $\sigma_\beta$ |
|---|---|---|---|---|---|---|---|
| 8 | 600 | 54 | 68 | 181,900 | 37.9 | | |
| 4 | | | | 93,214 | 38.8 | | |
| 2 | | | | 48,163 | 40.1 | | |
| 1 | | | | 26,828 | 44.7 | 40.3 | 2.6 |

time limit is again reached, but now the solution reported is within 7% of the theoretical optimal value as shown in Table 5.7. Again, the realization of this theoretical value may not exist due to the precedence-constraint relationship of the simulation system.

Another VHDL circuit used for search performance analysis is a 4-bit VHDL adder circuit which consists of 20 VET tasks each requiring 4 iterations. Using parallel method 3 and limiting

Table 5.7. Search performance to schedule a carry-lookahead adder onto 4 processors.

| Processors Used | Search Time (sec) | Lower Bound | Schedule Length | Search Nodes Generated | $\beta$ | $\mu_\beta$ | $\sigma_\beta$ |
|---|---|---|---|---|---|---|---|
| 8 | 600 | 108 | 116 | 229,682 | 47.8 | | |
| 4 | | | | 114,690 | 47.8 | | |
| 2 | | | | 57,258 | 47.7 | | |
| 1 | | | | 28,761 | 47.9 | 47.8 | 0.07 |

the search time to 600 seconds again provided the results shown in Table 5.8. When the input data is changed to schedule 4 processors instead of 8, the lower bound solution is immediately found as shown in Table 5.9. This optimal assignment establishes and optimal schedule length with the minimum number of processors. When interprocessor communication costs are considered, this schedule can outperform the 8 processor schedule as reported by [9] even though the schedule length is longer. The large difference between the costs of context switching for intraprocessor communications and sending messages accross the processor interconnection network for interprocessors communications benefit the seemingly inferior schedule. In general, an optimal mapping for a simulation system which minimizes the schedule lengths and the number of processors may perform better in the simulation run-time environment when interprocessor communications are realized. Comparing the reported schedule lengths with the lower bound values can guide the evaluation of these results.

The $\beta$ values for the VHDL simulations further show algorithm sensitivity to the simulation characteristics and not the run-time parameters when the rate of search node generation per processor, $\beta$, is examined. In Tables 5.6-5.9, the mean $\beta$ value may vary from simulation to simulation, but the standard deviation for these values is extremely low indicating a strong correlation to the individual simulation systems. The variation from data set to data set can be attributed to the implementation characteristics in making the actual task assignments and the combinatorics involved. Assigning a greater number of tasks at each search node generation point takes slightly more time

Table 5.8. Search performance to schedule a 4-bit adder onto 8 processors.

| Processors Used | Search Time (sec) | Lower Bound | Schedule Length | Search Nodes Generated | $\beta$ | $\mu_\beta$ | $\sigma_\beta$ |
|---|---|---|---|---|---|---|---|
| 8 | 600 | 14 | 16 | 241,537 | 50.3 | | |
| 4 | | | | 118,875 | 49.5 | | |
| 2 | | | | 59,659 | 49.7 | | |
| 1 | | | | 29,748 | 49.6 | 49.8 | 0.3 |

Table 5.9. Search performance to schedule a 4-bit adder onto 4 processors.

| Processors Used | Search Time (sec) | Lower Bound | Schedule Length | Search Nodes Generated | $\beta$ | $\mu_\beta$ | $\sigma_\beta$ |
|---|---|---|---|---|---|---|---|
| 8 | 1 | 28 | 28 | 361 | 45.1 | | |
| 4 | | | | 190 | 47.5 | | |
| 2 | | | | 99 | 49.5 | | |
| 1 | | | | 50 | 50 | 48.0 | 1.9 |

in this implementation reducing the search node generation rate per processor when comparing simulation systems which vary only in the number of processor upon which they are scheduled.

Prior to invoking the search process to find an optimal assignment of simulation tasks to processors, an estimate of the worst-case run-time can be helpful in understanding the complexity of the scheduling problem and setting an acceptable search time limit. Calculating the upper bound values for the size of the search space as described in Chapter 4 and executing short scheduling runs to extract a representable $\beta$ value for the given data, a worst-case search time can be calculated. For example, if 10 million possible search states exists, and the search node generation rate for the given data is 50 nodes/sec/processor, then applying 1024 processors to the problem can bound the search time at 195 seconds. If only 128 processors are used, the search time limit increases to 1,563 seconds or 26 minutes!

### *5.6 Summary*

Individual task systems are poorly behaved in relation to one another. Minor changes in their structure given the same number of tasks can result in vast changes in the size of the search space which must be implicitly explored to find an optimally minimum schedule. Therefore, validation of run-time performance for the parallel MBT algorithm requires analysis of scheduling performance when run-time parameters are modified. Varying the number of processors, adding upper bound communications, and incorporating load balancing activities among the working processor shows a significant change in overall performance. Applying more available processors to the search process, adding upper bound communications between the processors, and performing load balancing to maximize machine utilization provides near linear speed-up in the search process. Although parallel efficiency drops off slightly when more processors are added to the search process, the desired improvement in search time is realized. Unfortunately, given the combinatorics of optimally scheduling simulation systems, the necessary parallel search time may be to costly. To determine a worst-case search time cost, the scheduling process can be executed for a short time to obtain a

search node generation rate for the particular data set. This rate can then be used along with the number of available proccessors and the upper bound on the number of search states to determine the worst-case time allowance to find the optimal solution. If such times are determined to be unacceptable, then the scheduling process can be executed for a shorter period of time providing the 'best' solution for that given time. Also, the number of processors being scheduled can greatly affect the search time such that an optimal mapping may exist which minimizes the number of processors, the resulting search time, and the simulation run-time given the realization of interprocessor communication costs.

# *VI. Conclusions and Recommendations*

## *6.1 Conclusion*

The generalized scheduling problem with variable execution time (VET) tasks and feedback constraints is most difficult to solve. This problem, known to be $NP$-hard, requires exponential search time to solve when an optimal solution is desired. Unfortunately, computer memory limitations can easily be exceeded with such combinatoric search problems requiring a specialized approach to search space exploration. The MBT algorithm satisfies this requirement by maintaining search data only for the current branch of the search tree presently under investigation. The search process develops as a search tree and not a search graph since previously explored search states aren't maintained in order to conserve memory. Unfortunately, this results in the potential for multiple evaluations of the same search path – a necessary trade-off. When the algorithm is parallelized using a distributed-list search process with upper bound communications and load balancing activities, near linear speed-up is achieved.

The characteristics of the simulation system being scheduled onto a parallel, message passing computer play an important role in determining how the schedule generation process evolves. When the simulation system contains equal execution time (EET) tasks without feedback, list scheduling without anomaly compensation using the *all-iterations-first* strategy produces an optimal schedule within the minimum amount of search space and search time. With VET tasks and feedback, very large search spaces must be implicitly explored. Unfortunately, *a priori* analysis of the precedence-constraints can't produce a 'better' upper bound on the size of the total search space when compared to an independent task system. These systems cause the generation of unbalanced search trees and contain data specific combinatoric values at each stage of the assignment process. Also, the MBT algorithm can evaluate duplicate search paths further compounding the problem. The best upper bound on the search space remains an expression describing a non-uniform combination tree whose branch factor at each level of the tree is a combinatoric expression for EET task systems, and grows into a permutation expression for VET task systems. Unfortunately, the upper bound on the search space can't be reduced based upon the actual precedence-constraint characteristics of the simulation system being investigated.

Given a particular simulation system for scheduling, short trial runs can be executed to determine the node generation rate per processor of the parallel MBT algorithm on a given architecture. Using this value with the upper bound on the number of search states in the search space and the number of available processors, the worst-case execution time to find an optimal solution can be calculated. Based on this information, a prudent choice of an actual run-time limit can be imposed.

Should the process terminate when the time limit expires, the solution reported can't be guaranteed as the optimal solution. However, given the precedence-contraints of the simulation system, the solution may be optimal without certainty.

The validation process of the parallel MBT algorithm shows near linear speed-up for a generic iterative simulation with variable execution time tasks and feedback. The sensitive nature of the size of the search space when minor changes in the simulation structure/execution time requirements are made prohibits algorithm performance analysis based on variations of the input data. Validation of run-time performance for the parallel MBT algorithm requires analysis of scheduling performance when the run-time parameters are modified. Applying more available processors to the search process, adding upper bound communications between the processors, and performing load balancing to maximize machine utilization provides near linear speed-up in the search process. Also, the number of processors being scheduled can significantly affect not only the search time, but the simulation execution time as well. If the number of processors chosen for scheduling can result in a mapping of tasks to processors such that the schedule length is theoretically minimum based on bin packing and critical path analysis, and the search process finds such a schedule, the search process terminates immediately. No guarantee can be made as to where within the search space such a solution exists, but if found, the remaining search space can be ignored. Given the upper bound search state values for relatively small simulation systems, the time taken to check for such a solution is well worth the investment. Also, a schedule length which is longer than a schedule produced for a larger number of processors, may prove to perform better than the shorter schedule when the simulation system is executed given considerations for interprocessor communication costs.

In an iterative EET task system regardless of feedback contraints, the maximum number of tasks available for execution at any one time is the number of tasks in the original system (iterations not included). Therefore, if the number of processors is $\geq$ the number of original tasks, the task system can be optimally scheduled in $\gamma$ discrete step where $\gamma$ is the maximum depth of the search tree; *i.e.*, the number of tasks in the critical path including iterations. With at least as many available processors as ready tasks at each point in the scheduling process, only one scheduling combination can exist for each assignment. This results in an optimal schedule being produced in optimal time. However, the breadth of the schedule may be such that many tasks which communicate don't physically reside on the same processor. If interprocessor communication costs are again considered, such an optimal mapping may not prove to be optimal at simulation run-time.

An underlying assumption throughout this research has been that the number of task iterations is known prior to scheduling. This provides a boundary condition on the total number of possible search states since the MBT algorithm treats each task iteration as a new task; *e.g.*, a 10 task system requiring 10 iterations is logically treated as a 100 task system requiring 1 iteration for general simulation problems. Such a condition is necessary to ensure that a scheduling solution does exits. If the number of iterations were undefined or infinite, the algorithm would never run out of ready tasks to be scheduled and the process would continue infinitely. This doesn't apply when EET task systems are considered without feedback where the *all-iterations-first* scheduling decicion strategy reduces the number of logical tasks to the original amount regardless of the number of iterations: *e.g.*, a 10 task system requiring 10 iterations is still treated logically as a 10 task system.

This type of approach to iterative scheduling allows informed decisions to be made as to designing the task system structure, their communication and migration provisions, and the resulting assignment to the processors given the characteristics of the target machine. The number of iterations chosen for scheduling is largely independent of the actual number at run-time, especially with large and complex war gamming/VHDL simulations. Attempting to schedule a large number of iterations with the logical transform used by the parallel MBT algorithm can easily result in search time requirements for optimal solutions which exceed the life expectancy of the computer working on the problem. However, applying the scheduling algorithm incrementally to more and more task iterations may reveal scheduling patterns which identify the 'best' mapping of tasks to processors when considerations for task communication costs are made.

The Ada programming language proved to be beneficial in rapidly developing operational code for validating the search process. The developed algorithms mapped very easily into the structured design methodology of software development. Although code parallelization was difficult to validate on the *iPSC*/2 hypercube due to the limited parallel debugging environment, the additional code necessary for parallelization was minor. Initial serial code development of the MBT search algorithm using a Sun Sparc Station 2 and compiling on a Sun 4/490 minicomputer using Verdix Ada 6.0 was instrumental in the rapid code development obtained. The compilation time of the Sun 4/490 operating at 24 MIPS is approximately 24 times faster than the *iPSC*/2 SRM. Therefore, many small modifications to the initially developed code could be repeatedly made very quickly to validate the serial search process.

### *6.2 Recommendations*

Scheduling computer simulations onto a coarse-grain computer such as the *iPSC*/2 hypercube requires consideration of interprocessor and intraprocessor communication costs. When a task must

send a message to another task residing on the same processor, the message must be posted and the operating system must block the task and provide cpu time to the receiving task before the message can be received. Also, when a task must send a message to another task residing on a different processor, not only will time be spent in transmitting the message to the other processor, but the same 'process-swapping' cost may be incurred as well. These factors can have a significant affect on the run-time performance of the simulation.

The results of this investigation show that data decomposition of the search process using a parallel, message passing computer can achieve near linear speed-up in the time required to generate an optimally minimum schedule for execution on the same computer. Unfortunately, interprocessor and intraprocessor communication costs are assumed to be negligible causing no consequence of allowing task migrations which further minimize schedule lengths. Also, the largest parallel machine used to validate the search process contained only eight processors. Therefore, several expansions of this research are viable:

1. Consider interprocessor communication costs in developing optimally minimum schedules. Under the list scheduling approach, permutations of assignment possibilities must now be considered greatly increasing the search space for an optimal solution. If sub-optimal solution are acceptable, heuristic methods could be incorporated into the schedule generation process to help reduce the search space should it become prohibitively larger.
2. Consider applicability of task migrations; can this be effectively implemented on a hypercube such that the implementation costs don't exceed the schedule cost when task migration isn't allowed? If the overhead involved in migrating tasks or invoking copies of tasks on different processors based on current loading conditions of the parallel computer exceeds the cost of the excessive interprocessor communications, then such an approach may not be viable. However, if the simulation system can be designed so that the tasks are coarse-grain in nature in relation to their intertask communication costs, this approach may prove very fruitful.
3. Modify the implementation code to the C programming language for portability onto larger hypercubes such as *iPSC/860* or the *iPSC/Paragon* supercomputer for more thorough analysis of parallelized performance and speed-up capabilities.
4. Modify the implementation code to run on a shared memory architecture such as the Connection Machine. This approach may reduce the communications cost associated with distributed memory architectures providing greater spead-up potential.

Parallel simulations take full advantage of current computer technology in acquiring design data for future computer/electronic systems. Utilizing the computing power of such machines

requires careful and deliberate mappings of the simulation systems onto the available processors in order to minimize the overall execution time of the simulation through many iterations. Without optimal mappings of simulation tasks onto processors, much time can be wasted in obtaining the desired data. Simulating battle field scenarios prior to actual conflicts can not only guide the development and deployment of weapon systems, but save the lives of many combatants challenged to defend their country in the face of an aggressive and hostile force. Also, the development of VLSI circuits using the hardware description language VHDL, and the exercising of these simulations on large, parallel computers is an important tool in bringing functionally sound circuits to life in the minimum amount of time.

# Appendix A. *Code Structure*

## *A.1 Structure Chart*

The MBT algorithm was developed using the structured design methodology. In that endeavor, the following structure chart describes the implemented modules and their relationships to one another.

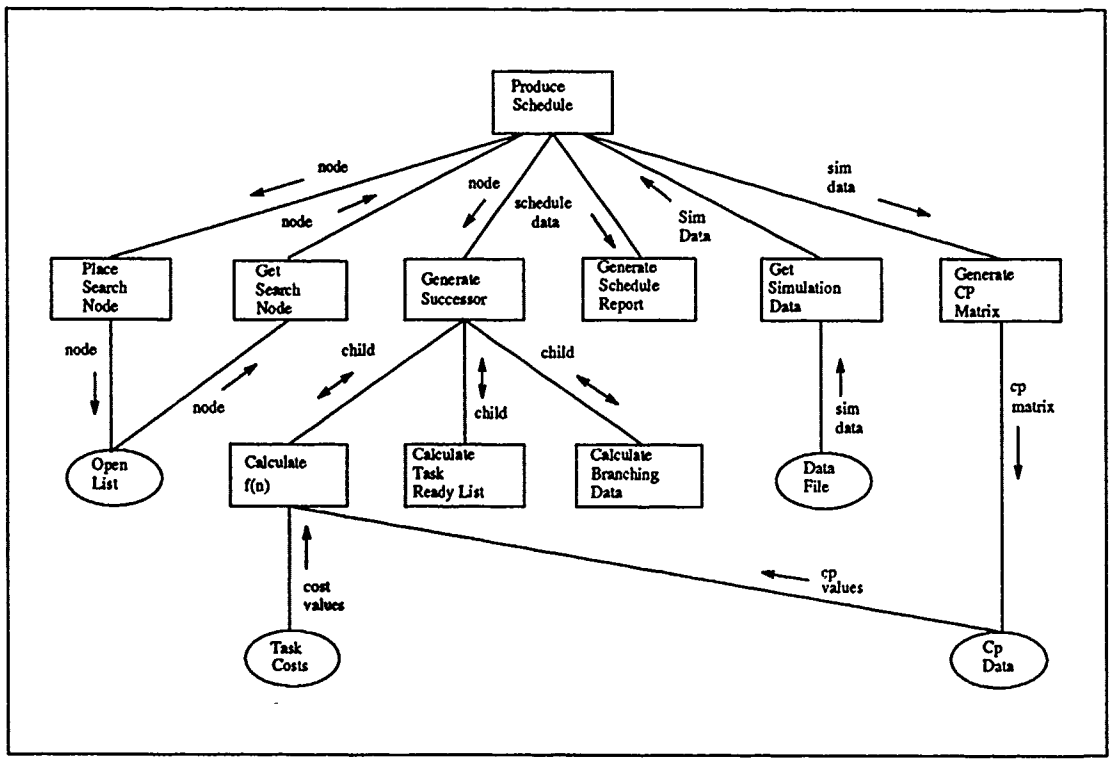



Figure A.1. Structure chart for MBT search software.

## *A.2 Functional Programming*

Simulation systems exhibit characteristics which can be used to change the schedule generation process by reducing the search time. The difference exists in the manner in which successor search nodes are generated. For example, under the *all-iterations-first* strategy, when a task is assigned to a processor, all iterations can be assigned at once. Therefore, two separate Ada packages were developed. The first package contains all the procedures for maintaining the OPEN list and producing the results. The second package contains the procedures for generating the

successor search nodes. The procedures within this package vary depending upon what type of simulation system is being scheduled. These packages are then included into the main procedure which performs the parallel MBT algorithm steps.

The developed code was documented using the following documentation standards for each procedure and function:

- Date
- Procedure/Function name
- Description
- Algorithm
- Modules called
- Order-of analysis
- History

# Appendix B. *User Manual*

## *B.1 Input Data Format*

Executing the parallel MBT search program on the *iPSC/2* can be done with the user interactively entering the description of the iterative task system or by building the data file prior to execution. The input data file consists of the number of tasks, iterations, and processors to be scheduled, the adjacency matrix, and the set of task execution costs. Each data item must be separated by a blank line with an input order as follows:

1. number of tasks
2. number of iterations
3. number of processors to be scheduled
4. adjacency matrix
5. task costs

The following input data example reflects the scheduling requirement for the simulation system shown in Figure B.1.

```
  7  <== Number of Tasks

  2  <== Number of Iterations

  2  <== Number of Processors

0001000   <== Adjacency Matrix
0001000
0000100
0000010
0000011
0000000
0000000

2315342   <== Task Costs

(commentary information can be added here!)
```

Data files can be easily built by using the interactive mode of the program. Upon completion of data input, the user is prompted to either write the data to disk or immediately begin the search. The following script is an example of this procedure which builds the data file shown:

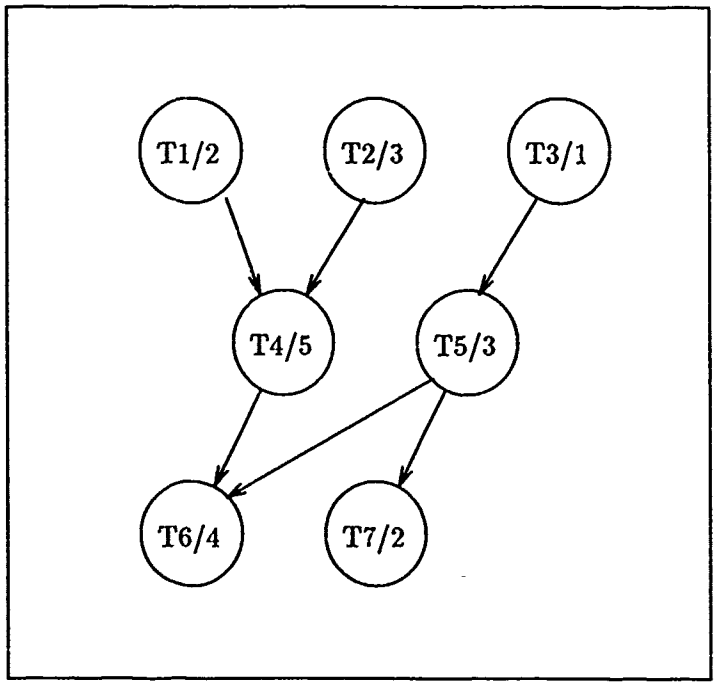



Figure B.1. A generic simulation system.

```
Enter input filename or <CR> for manual entry:
Enter number of tasks in the graph ==> 3

Enter number of task iterations to schedule ==> 3

Enter number of processors to schedule ==> 3

Enter execution cost of task   1 ==> 2

Enter number of arcs for task  1 (1-100) ==> 1

    Arc  1 of task  1 goes to task ==> 2

Enter execution cost of task   2 ==> 2

Enter number of arcs for task  2 (1-100) ==> 1

    Arc  1 of task  2 goes to task ==> 3

Enter execution cost of task   3 ==> 4

Enter number of arcs for task  3 (1-100) ==> 0

Save user input to disk [Y/N]? y
```

```
Saving input to 'task_in.dat'

Enter the name of your output file (default is task.dat) => new_out.dat
```

The resulting data file 'task_in.dat' saved to disk follows the input data format:

```
      :
  3   <== Number of Tasks

  3   <== Number of Iterations

  3   <== Number of Processors

010   <== Adjacency Matrix
001     '
000     .

224   <== Task Costs
      :
```

### *B.2 Parallel Execution*

To invoke the parallel MBT search program, at least one node of the *iPSC/2* hypercube must be allocated. At the system prompt, type 'host' to invoke the process. Since three versions of the parallel search process were developed to determine the method which best utilized the available processing power and obtained the most speed-up, the user must specify which node program the 'host' program will load and run. The three node programs are (1) nodebts1, (2) nodebts2, and (3) nodebts3. Node program nodebts1 performs parallel search method 1, nodebts2 performs parallel search method 2, and nodebts3, the 'best' program, performs parallel search method 3 as outlined in chapter 4. A script example of the process to load and run parallel search method 3 follows:

```
Enter node filename to load: nodebts3
loading nodebts3 program onto 8 processors

*******************************************************
*        The Parallel MBT Task Scheduler              *
*******************************************************
Enter input filename or <CR> for manual entry: task_in65.dat

Enter the name of your output file (default is task.dat) => test_out.dat
Enter search time limit in seconds: 1800


Working!!!
```

If only 1 processor is allocated when the search program is invoked, the 'host' program automatically loads nodebts1 onto the processor because using only one node requires no interprocessor communications.

### *B.3 Output Data Format*

The present implementation of the parallel MBT search algorithm produces the following output given the simulation system of Figure B.1:

```
Node File: nodebts3
Adjacency Matrix:
      1  2  3  4  5  6  7
    ....................
 1 :  0  0  0  1  0  0  0
 2 :  0  0  0  1  0  0  0
 3 :  0  0  0  0  1  0  0
 4 :  0  0  0  0  0  1  0
 5 :  0  0  0  0  0  1  1
 6 :  0  0  0  0  0  0  0
 7 :  0  0  0  0  0  0  0

# Task Dependencies in Graph :    6
Task Costs:
      1  2  3  4  5  6  7
    ....................
      2  3  1  5  3  4  2

Critical Path Values:
(row = iteration, column = task starting point
      1  2  3  4  5  6  7
    ....................
 1 : 16 17 12 14 11  8  4
 2 : 11 12  8  9  7  4  2

Number of Iterations =  2
Number of processors to schedule =  2
Number of processors used in the search process =   8
# search nodes created:  626
# search nodes requested for load balancing:  8
# search nodes sent to satisfy a load balancing request:  5
# bounding msgs:  1
lower bound:  20 Time Units
node run times (sec):
    1.48
    1.43
    1.53
    1.47
```

```
    1.47
    1.45
    1.43
    1.42
Optimal Schedule:
 P 1 | P 2 |   Time
-------------   ----
    1|    3|    1
    1|    2|    2
    1|    2|    3
    1|    2|    4
    3|    4|    5
    5|    4|    6
    5|    4|    7
    5|    4|    8
    2|    4|    9
    2|    6|   10
    2|    6|   11
    4|    6|   12
    4|    6|   13
    4|    5|   14
    4|    5|   15
    4|    5|   16
    6|    7|   17
    6|    7|   18
    6|    7|   19
    6|    7|   20


With Cost :   20 Time Units
Parallel Search Time:    10 Seconds
```

## *Bibliography*


1. Aki, Selim G. *The Design and Analysis of Parallel Algorithms*. New Jersey: Prentice-Hall, Inc., 1989.
2. Anger, Frank D. *et al.* "Scheduling with Sufficient Loosely Coupled Processors," *Journal of Parallel and Distributed Computing*, *9*:87–92 (1990).
3. Beard, R. A. and Gary B. Lamont. "Determination of Algorithm Parallelism in NP-Complete Problems for Distributed Architectures." *Proceedings of the Fifth Distributed Memory Computing ConferenceI*. 42–51. April 1990.
4. Bernhardt, Mike *et al.* "1990 Gordan Bell Prize/IS-436." 1990 Gordon Bell Prize awarded to scientists using Intel iPSC/860 parallel supercomputer.
5. Booch, Grady. *Software Components with Ada*. Menlo Park, CA: The Benjamin/Cummings Publishing Company, Inc., 1987.
6. Casavant, Thomas L. and Jon G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, *14*(2) (February 1988).
7. Christofides, Nicos. *Graph Theory, An Algorithmic Approach*. London: Academic Press, 1975.
8. Coffman, Edward G. *et al. Computer & Job/Shop Scheduling Theory*. New York: Wiley & Sons, Inc., 1976.
9. Comeau, Ron. *Transforming VHDL Circuit Designs for Parallel Simulation*. MS thesis, AFIT/GCS/ENG/91D-09, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
10. Cormen, Thomas H. *et al. Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1991.
11. Corporation, Intel. *Parallel Program Primer*. Intel Corporation.
12. Cvetanovic, Z. and C. Nofsinger. "Parallel Astar Search on Message-Passing Architectures," *Hawaii International Conference on System Sciences*, 82–90 (1990).
13. DeCegama, Angel L. *The Technology of Parallel Processing*. New Jersey: Prentice-Hall, Inc., 1989.
14. Gait, Jason. "Scheduling and Process Migration in Partitioned Multiprocessors," *Journal of Parallel and Distributed Computing*, *8*:274–279 (1990).
15. Gendreau, Thomas B. "Scheduling in Distributed Systems." *Proceedings from the Second Workshop on Large-Grained Parallelism*. 34–36. November 1987.
16. Gustafson, John L. et.al. "Development of Parallel Methods for a 1024-Processor Hypercube," *SIAM Journal of Science and Statistical Computing*, *9*(4):609–638 (July 1988).
17. Hayes, John P. and Trevor Mudge. "Hypercube Supercomputers," *Proceedings of the IEEE*, *77*(12):1829–1841 (1989).
18. HUBB systems, Inc. *Computing Strategies for SDI Battle Management Command, Control, and Communications Assignment Algorithms*. Final Report, Huntsville, AL: HUBB Systems, Inc., January 1988 (AD-B121 838).
19. Kasahara, Hironori and Seinosuke Narita. "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, *11*(c-33):1023–1029 (November 1984).

20. Kasahara, Hironori and Seinosuke Narita. "An Approach to Supercomputing using Multiprocessor Scheduling Algorithms." *Proceedings of the First International Conference on Supercomputing Systems*. December 1985.

21. Klappholz, David and Haeng-Chul Park. "Parallel Process Scheduling for a Tightly-Coupled MIMD Machine." *Proceedings of the 1984 International Conference on Parallel Processing*. 315–321. August 1984.

22. Korf, Richard E. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, *27*:97–109 (1985).

23. Lamont, Gary B. et. al. *Compendium of Parallel Programs*. Electrical Engineering Department, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990. parallel program implementations.

24. Lee, Chung-Yee *et al.* "Multiprocessor Scheduling with Interprocessor Communication Delays," *Operations Research Letters*, 7(3):141–147 (June 88).

25. Li, Keqin and Kam Hoi Cheng. "Static Job Scheduling in Partitional Mesh Connected Systems," *Journal of Parallel and Distributed Computing*, *10*:152–159 (1990).

26. Liu, Joseph W.H. *Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization*. Final Report, Oak Ridge, TN: Oak Ridge National Laboratory, October 1987 (AD-A187 038).

27. Mayr, Ernst. *Well Structure Parallel Programs are not Easier to Schedule*. Technical Report, Department of Computer Science, Stanford University CA: Stanford University, September 1981 (AD-A113 400).

28. McNear, Andrew E. and Guy R. Booth. "The Mesh Connection Network and *iPSC*/2 Performance Analysis using Ada." CSCE656 Project, March 1991.

29. Molloy, Michael K. "Requirements for the Performance Evaluation of Parallel Systems." *Proceedings from the Second Workshop on Large-Grained Parallelism*. 63–65. November 1987.

30. Papadimitriou, C. and M. Yannakakis. "Scheduling Interval-Ordered Tasks," *SIAM Journal of Computing*, *8*:405–409 (1979).

31. Pearl, Judea. *Heuristics, Intelligent Search Strategies for Computer Problem Solving*. New York: Addison-Wesley Publishing Company, 1985.

32. Pramanick, Ira and Jon G. Kuhl. "Study of an Inherently Parallel Heuristic Technique." *Proceedings of the International Conference on Parallel ProcessingIII*. 95–99. August 1991.

33. Prasanna, Srinivasa G. and Bruce R. Musicus. "Generalised Multiprocess Scheduling Using Optimal Control." *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*. 216–228. New York NY: ACM Press, July 1991.

34. Price, Camille, C. and S. Krishnaprasad. "software Allocation Models for Distributed Computing Systems." *The 4th International Conference on Distributed Computing Systems*. 40–48. 1984.

35. Ramamritham, Krithivasan and John A. Stankovic. "Dynamic Task Scheduling in Distributed Hard Real-Time Systems." *The 4th International Conference on Distributed Computing Systems*. 96–107. 1984.

36. Rudolph, Larry *et al.* "A Simple Load Balancing Scheme for Task Allocation in Parallel Machines." *3rd Annual ACM Symposium on Parallel Algorithms and Architectures*. 237–245. New York NY: ACM Press, July 1991.

37. Sahni, Sartaj. "Scheduling Multipipeline and Multiprocessor Computers." *Proceedings of the 1984 International Conference on Parallel Processing*. 333–337. August 1984.

38. Sartor, JoAnn M. *Optimal Iterative Task Scheduling for Parallel Simulations*. MS thesis, AFIT/GCS/ENG/91M-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991.

39. Sartor, JoAnn M. *et al.* "Mapping Precedence-Constrained Simulation Tasks for a Parallel Environment." *Proceedings of the Sixth Distributed Memory Computing Conference*. 2–10. May 1991.

40. Stankovic, John A. "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Transactions on Computers*, *C-34*(2):117–130 (February 1985).

41. Stankovic, John A. and Inderjit S. Sidhu. "Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups." *The 4th International Conference on Distributed Computing Systems*. 49–59. 1984.

42. Taylor, Paul J. *Requirements Analysis for a Hardware, Discrete-Event, Simulation Engine Accelerator*. MS thesis, AFIT/GCE/ENG/91D-11, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.