MULTIPROCESSOR SYSTEM EVALUATON AND PROGRAMMING ENVIRONMENT

Milos D. Ercegovac
Final Report

April 1986
CSD-860066

UNIVERSITY OF CALIFORNIA
LOS ANGELES
COMPUTER SCIENCE DEPARTMENT

Sandia National Laboratories
Contract No.25-3074

"Multiprocessor System Evaluation and Programming Environment"

- Final Report -

Principal Investigator: Prof. M.D. Ercegovac,

Associated Faculty: Prof. T. Lang,
Prof. R.M. Muntz,

Graduate Research Assistants: P.K. Chan,
T.M. Ravi

April 1, 1986
Los Angeles

This is the final report summarizing the research and development performed under the Sandia National Laboratories' Contract No. 25-3074 "Multiprocessor System Evaluation and Programming Environment". It consists of four parts:

Part 1 describes a multiprocessor simulator.

Part 2 discusses allocation of tasks in a data-driven multiprocessor and SANDAC IV System.

Part 3 and 4 contain the corresponding software on magnetic tapes.

The main contributions in the design of the simulator were made by P.K. Chan and in the allocation of tasks by T.M. Ravi. The support of the Sandia National Laboratories and technical cooporation with its technical staff members George Davidson and Paul E. Pierce are greatly appreciated.

Part I

SANDAC Multiprocessor Simulation

University of California, Los Angeles
Computer Science Department

SANDAC Multiprocessor Simulation: Final Report

P. K. Chan and M. D. Ercegovac

Abstract - Sandy, a simulator for the SANDAC [2] multiprocessor has been developed. This simulator is capable of simulating the multi-tasking and message passing inter-processor communication environments in SANDAC. Inputs to sandy are the C source codes that a user actually would run on SANDAC. Sandy provides performance estimates which makes it a viable tool in the development phase of projects to run on SANDAC. This document examines the organization of sandy.

1. Introduction

This document describes the organization of sandy and examines briefly its source code. It is written for those who intend to adapt sandy for a different architecture other than SANDAC, or to extend and refine the current capabilities of the simulator.

Simulation is one of the many ways to abstract, characterize and study a complex system. The levels of abstraction can be microscopic, or macroscopic, or somewhere in between. Choosing an appropriate level of abstraction requires a judicious choice between the level of confidence that one would like to acquire from simulation and the constraints of resources. Before a simulator is operational, it is difficult to predict or quantify the level of confidence. For a given level of confidence, it is not easy either to decide on the appropriate level of abstraction to satisfy the need. These problems remain some of greatest challenges in the area of simulation methodology. Study and evaluation

of a complex system through simulation traditionally involves feeding a representative set of inputs to the simulation model and evaluating the outcome.

The simulator `sandy` is different from stochastic simulators such as `PAWS` or `CACI NETWORK II.5` [3]. Those are stochastic simulators in the sense that their simulation models are stochastic: a user has to present probabilistic parameters to the simulator. `Sandy` uses a deterministic model of the system to be simulated.

Inputs to `sandy` are C source code files. To avoid confusion, we refer to the C source codes input to the simulator as *input source codes*; whereas the source codes of the simulator itself is referred to as *source codes*.

Some of the early suggestions was to build a simulation tool to simulate the SANDAC multiprocessor at the machine instruction level; this means that a simulator would interpret the assembly code generated by the compiler. This scheme would make a simulator highly machine dependent, since most of the microscopic system attributes would have to be mimicked quite rigidly. The other suggestion was to simulate SANDAC at the C programming language level. At this level, the simulator could interpret directly the input source code. System attributes subsequently would be abstracted at a macroscopic level. As far as the level of confidence is concerned, the first approach is undisputely the better one. One weakness of the second approach is worth noting however, since the simulator is abstracting system attributes at a higher level, detailed machine attributes and compiler optimization issues (such as register allocation, and code optimization) are not taken into account. Consequently, this affects the accuracy of the simulation. Nevertheless, we adopted the second scheme due to concerns of complexity, flexibility, and availability of resources.

The simulator comprises three major components: 1. a parser, 2. an interpreter, and 3. an event scheduler. It is our intent that this organization effectively divides the simulator into machine independent and machine dependent parts. The parser and the interpreter are essentially machine-independent, whereas the event scheduler is largely machine dependent.

The parser is a modified version of the first pass of the Portable C Compiler (PCC) [5] for RISC. The parser performs lexical and syntactic checking on an input C source program. It also generates an intermediate file containing parse trees to be interpreted by the interpreter. A small part of the interpreter consists of the modified second pass of the PCC [7]. This part reads intermediate files and converts the parse trees into an internal form - expression trees. The rest of the interpreter is an evaluator which does the interpretation (execution) of the expression trees. Each *step* in interpreting an expression tree is considered by the event scheduler as an event. One of the functions of the event scheduler is to evoke and schedule an event according to the number of machine cycles which is required by the SANDAC multiprocessor to execute the *step*. Section 3.3 will elaborate on this point. Other functions of the event scheduler include maintaining resources (e.g., global bus, global memory and local message queues), and interlocks among tasks in an unambiguous fashion.

The faithfulness of the simulator in mimicking the real machine depends on two facts. First, it relies on the accuracy of the timing parameters which are supplied by a user to depict execution times of the expression trees. Secondly, since it is extremely difficult to simulate interrupts in all VRTX (a registered trademark of Hunter & Ready Inc.) and standard C library calls [1], the simulator simply requires that all system calls be non-interruptible. In simple words, the simulation of interrupts is imprecise. Section 3.3 will elaborate on this issue.

The size and speed of the simulator have always been a concern, and several changes have been made during development to improve these aspects. It is our belief that the current implementation of the simulator does maintain a balance between these conflicting issues.

The current version of the parser consists of more than 10000 lines of code, and for the interpreter and event-scheduler together, occupy roughly 4000 lines of code.

1.1. How to use it

We illustrate with an example the steps to run a simulation. Suppose we have five tasks. To run the parser, first we parse each task source code into parse trees

```
cc -E task11.c | pas > ptree.11
cc -E task12.c | pas > ptree.12
cc -E task13.c | pas > ptree.13
cc -E task21.c | pas > ptree.21
cc -E task22.c | pas > ptree.22
```

The reason for saying 'cc -E' is to get rid of the comments and to expand macro definitions in the task files. To run the simulator on the parse trees, use,

```
sandy -2 ptree.11 ptree.12 ptree.21 ptree.22 ptree.31
```

or, simply

```
sandy -2 ptree.*
```

in csh; which means that you have two processors in the system. The tasks are assigned to the processors as,

processor 1 : task11 task21 task31
processor 2 : task12 task22

with task11 having the highest priority w.r.t processor 1, task21 has the next highest and so on. The rules for allocating tasks to processor are described in Section 4.2. The maximum number of processors is 16 in this installation. This is controlled by a parameter MACHINE defined in the header file "../ran/h.sim". It doesn't matter if the number of tasks is less than the number of machines, some of the processors would simply be idle.

Notice that we have restricted each task to be self-contained in a file. The main function of each task *has* to be named `main()`.

2. The Parser

This parser is modified from the first pass of the Portable C compiler that was originally designed by Steve C. Johnson. Therefore, the style of presentation in this section is much influenced by Steve C. Johnson's article [5].

This parser does lexical analysis, parsing, symbol table maintenance, and parse tree building. The source code for the Parser exists as a set of files in directory "`../firstpass/`", namely,

```
cgram.c  cgram.y  code.c  comm1.c common local.c
local2.c macdefs manifest mfile1 newdope optim.c
pftn.c reader.c rodata.c scan.c trees.c xdefs.c.
```

2.1. The Source Files

Two files, "`manifest`" and "`macdefs`", are header files included with all other files. "`Manifest`" has declarations for the node numbers, types, storage classes, and other global data definitions. "`Manifest`" has size and alignment of various data representations, and for the purpose of simulations all word sizes are set to be `16`, which is the word length of the SANDAC multiprocessor. Two other files, "`mfile1`" and "`mfile2`", contain the data structure (e.g., `NODE`) and manifest definitions for the first pass.

There is a file, "common", containing routines used in pass one and two. These include routines for allocating and freeing trees, walking over trees, printing debugging information, and printing error messages (e.g., `cerror()`).

The first pass is obtained by compiling and loading `scan.c`, `cgram.c`, `xdefs.c`, `pftn.c`, `trees.c`, `optim.c`, `local.c`, `code.c`, and `comm1.c`. `Scan.c` is the lexical analyzer, which is used by `cgram.c`, the result of applying `yacc` to the input grammar `gram.y`. `Xdefs.c` is a short file of external definitions. `Pftn.c` maintains the symbol table, and does initialization. `Trees.c` builds the expression trees, and computes the node types. `Optim.c` does some optimization on the expression trees. `Comm1.c` includes "common", which contains service routines common to the two passes of the compiler. The files `local.c` and `code.c` contain RISC dependent code for generating subroutine prologs, switch code, and the like. Modifications have been made on the files `pftn.c`, `local.c`, and `code.c` for the purpose of realizing interpretation.

2.2. Intermediate File Format and Nodes

Parse trees produced by the parser are written to the standard output. We recommend the user to redirect the output to an intermediate file. The intermediate file is a text file organized into lines. The parse trees are presented in Polish Prefix form: first there is a line beginning with a period '.', followed by the source line number and the source name on which the expression appeared. The successive lines represent the nodes of the parse tree, one node per line. Each node contains the node number, type, and any values (e.g., values of constants) that may appear in the node. A node on a parse tree can either be an operator or an operand. Lines representing nodes with descendants are immediately followed by the left subtress of descendants, then the right. Since the number of descendants of any node is completely determined by the type of node, there is no need to mark the end of the tree.

There are four other line types in the intermediate file. Lines beginning with a left square bracket '[' represent the beginning of blocks, lines beginning with a right square bracket ']' represent the end of blocks. The remainder of these lines tell how much space, and how many register variables, are currently in use.

Lines beginning with a right bracket ')' represent the beginning of a function. The name of the function immediately follows the bracket.

Lines beginning with the letter 'L' represent a label, which can be used either for carrying a constant or marking a location.

Lines beginning with a tab character '^T' represent a assembly instruction generated by the parser, some of them are relevant to the interpreter and the rest being just assembly pseudo codes.

Nodes which share a common characteristic are grouped. The file "newdope" contains a complete list of nodes. The nodes on the expression trees are classified into three types, 1) leaf node (LTYPE): nodes belong to this class have no descendant, 2) unary node (UTYPE): these nodes are unary operators which carry a single left descendant; and 3) binary node (BITYPE): these nodes are binary operators which carry two descendants.

A good understanding of the semantics of these nodes is vital to the accuracy of simulation. Section 6 describes how one would assign time values to these nodes to affect timing in the simulation. In Table 1. we provide a list of nodes for the purpose of further discussion.

Of the five leaf nodes,

1. REG: denotes a variable which is found in a register.

2. NAME: denotes a variable whose name is given.

3. OREG: denotes a variable whose address can be computed as the sum of a con-

stant (offset or base address) plus the contents of a register.

4. ICODES: denotes an immediate constant, which may be an explicit constant or an address constant, or a combination of these, depending on the context.

5. CCODES: denotes a set a binary values which is encoded in the processor status word.

Of the fourteen unary nodes, FORCE is used to ensure that a result must occur in a particular register (e.g., the return register). STARG denotes a structure argument. UNARY MUL is the indirection operator. SCONV, PCONV are operators denoting "shape conversion" and "pointer conversion", respectively.

The majority of the remaining nodes are binary. One should find a close correspondence with the binary operators appearing in the C programming language [6] and these nodes, except for a few. RISCOR is for indirection to a register variable. CM is used in the concatenation of arguments in function calls. QUEST is the operator for conditional expressions. CBRANCH is a branch conditioned on the evaluation of the left subtree. ULE, ULT UGE, etc., are the unconditional branches.

There are some special nodes which are not found in the PCC [5]. They are invented just for the purpose of facilitating interpretation. They are: RMSTACK relinquishes a stack before returning to the previous context, GESTACK acquires new stack space due to change to a new context, TOSTK accesses data from the stack, RSUB subtracts two registers from one another, RADD adds two registers together, SW is a switch operation to jump to different locations, RET is return to the caller, and JMP sets the program counter.

2.3 Operand Types

The file "manifest" defines sixteen basic operand types for the intermediate operators. The important basic types and type modifiers to notice for the interpreter are:

1. CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, STRTY, UNIONTY denote character, short integer, integer, long integer, floating point, double precision, structure, and union type, respectively.

2. PTR: pointer type modifier.

3. FTN: function type modifier.

4. ARY: array type modifier.

Different complex operand types can be derived by combining the basic types and modifiers. For instance, when combining ARY and FLOAT we have a floating-point array.

2.4. What is Done and What is Not ?

Most of the nodes declared in the file "newdope" are recognized by the interpreter. There are three special nodes STARG, STCALL and UNARY STCALL, which are known respectively as, structure argument to a function call, calls of a function with nonzero, and zero arguments. They are not defined in the standard C language and hence are not yet implemented in the current installation (STASG the structure assignment operator is implemented). Extension to include those operations, however, is possible.

Token	Operation	Type	Token	Operation	Type
NAME	"NAME"	LTYPE	REG	"REG"	LTYPE
OREG	"OREG"	LTYPE	ICON	"ICON"	LTYPE
CCODES	"CCODES"	LTYPE	UNARY MUL	"U*"	UTYPE
UNARY MINUS	"U-"	UTYPE	UNARY CALL	"UCALL"	UTYPE
UNARY FORTCALL	"UFCALL"	UTYPE	NOT	"!"	UTYPE
COMPL	"~"	UTYPE	FORCE	"FORCE"	UTYPE
RVAL	"RVAL"	UTYPE	INIT	"INIT"	UTYPE
SCONV	"SCONV"	UTYPE	PCONV	"PCONV"	UTYPE
RISCOR	"riscor"	BITYPE	PLUS	"+"	BITYPE
ASG PLUS	"+="	BITYPE	MINUS	"-"	BITYPE
ASG MINUS	"-="	BITYPE	MUL	"*"	BITYPE
ASG MUL	"*="	BITYPE	AND	"&"	BITYPE
ASG AND	"&="	BITYPE	QUEST	"?"	BITYPE
COLON	":"	BITYPE	ANDAND	"&&"	BITYPE
OROR	"||"	BITYPE	CM	","	BITYPE
COMOP	",OP"	BITYPE	ASSIGN	"="	BITYPE
DIV	"/"	BITYPE	ASG DIV	"/="	BITYPE
MOD	"%"	BITYPE	ASG MOD	"%="	BITYPE
LS	"<<"	BITYPE	ASG LS	"<<="	BITYPE
RS	">>"	BITYPE	ASG RS	">>="	BITYPE
OR	"|"	BITYPE	ASG OR	"|="	BITYPE
ER	"^"	BITYPE	ASG ER	"^="	BITYPE
INCR	"++"	BITYPE	DECR	"--"	BITYPE
STREF	"->"	BITYPE	CALL	"CALL"	BITYPE
FORTCALL	"FCALL"	BITYPE	EQ	"=="	BITYPE
NE	"!="	BITYPE	LE	"<="	BITYPE
LT	"<"	BITYPE	GE	">"	BITYPE
GT	">"	BITYPE	UGT	"UGT"	BITYPE
UGE	"UGE"	BITYPE	ULT	"ULT"	BITYPE
ULE	"ULE"	BITYPE	ARS	"A>>"	BITYPE
LB	"["	BITYPE	CBRANCH	"CBRANCH"	BITYPE
FLD	"FLD"	UTYPE	PMCONV	"PMCONV"	BITYPE
PVCONV	"PVCONV"	BITYPE	RETURN	"RETURN"	BITYPE
CAST	"CAST"	BITYPE	GOTO	"GOTO"	UTYPE
STASG	"STASG"	BITYPE	STARG	"STARG"	UTYPE
STCALL	"STCALL"	BITYPE	UNARY STCALL	"USTCALL"	UTYPE
JMP	"jmp"	UTYPE	RET	"ret"	UTYPE
RADD	"radd"	LTYPE	RSUB	"rsub"	LTYPE
SW	"sw"	UTYPE	RSI	"rsubi"	UTYPE
JEQ	"jmpeq"	UTYPE	JGT	"jmpgt"	UTYPE
TOSTK	"tostk"	LTYPE	GESTACK	"getstk"	LTYPE
RMSTACK	"rmstk"	LTYPE			

Table 1. A List of Nodes

3. The Interpreter

The source codes for the interpreter exist as a set of files in the directory "`../sandia/`". The heart of the interpreter is the function `evaluate()`. This function traverses an expression tree non-recursively, and performs actual calculations on the operands. The main challenge in designing the interpreter is the requirement that the interpreter and the event scheduler cooperate as co-routines. Since the C program language doesn't support co-routines, in order to simulate concurrent tasks, the inter-

preter has to be written such that it interprets an input source program in a stop-and-go (re-entrant) manner. For instance, the interpreter, while in the middle of traversing an expression tree of a task, is able to transfer control to the event scheduler for it to initiate the interpretation of the expression tree of another task.

3.1. Data types

The complete set (size of 16) of basic operand types recognizable by the parser is defined in "`manifest`". As far as the interpreter is concerned, the union structure UVAL defines the legitimate basic data types recognizable by the interpreter. This union structure is declared in the header file "`../firstpass/defile1`". Currently, the following *basic* data types are implemented in the interpreter: `float`, `int`, `double`,, `char`, `long`, and `short`. `Long` and `short` data types are treated by the interpreter as `integer` data type.

The precision in the computation of arithmetic operations depends on the machine where `sandy` is installed.

3.2. The Source Files

The following three files comprises the principal source code of the interpreter, namely, `eval.c`, `sys.c`, and `misc.c`.

1. The file `eval.c`, as the name suggests, performs evaluation of the expression trees. The procedure `evaluate()` directs the control flow in the process of expression tree traversals. `Call()` handles initialization of stack space and register allocations. `Eval_ltype()`, `eval_btype()`, and `eval_utype()` perform evaluation on the leaf nodes, binary nodes, and unary nodes respectively.

2. The file `sys.c` contains a) all mathematical functions as defined in the standard C math-library, b) four VRTX [1] message queueing and dequeueing functions, and c) some standard i/o functions (see also section 4.2). Before one would introduce new library functions into the interpreter, one should *declare* the new function with the function `initial()` in the file "`sys.c`, then *define* the new function in this file.

3. The file `misc.c` contains miscellaneous functions for maintaining symbols (e.g., variables, labels, integer constants, floating-point constants, string constants, and function names) that appear in the intermediate file.

3.3. The Function `evaluate()`

The backbone of this function is a non-recursive expression tree traversal algorithm. All expression trees are traversed top-down from the root. For unary node, there is only one way to go, down. For binary node, one may visit either the left or the right descendent first, the order of traversal is determined by the operation of the node. For instance, binary nodes `ASSIGN` and `CALL` required a left postorder traversal, whereas the nodes `PLUS` and `INCR` have to be traversed in the reverse order.

An algorithm which requires two stacks and one visit count is invented to serve the purpose. Stacks are used to make the function re-entrant. A data stack (`d_stack`) is used for holding intermediate results that appear on the arcs of an expression tree. This data stack is also used for transmitting arguments in function calls. Functions which are used for manipulating the data stack are: `pop_d()` and `push_d()`. An operator stack (`p_stack`) is used to keep track of the operators which are "partially" interpreted. Functions which are related to the data stack are: `pop_p()` and `push_p()`. The visit count is used to keep track of the number of descendants that have been visited. For instance, when the visit count becomes 3 for a binary node, this means that its two operands are available on the data stack, and hence the node can be fired. Some nodes such as `INCR` and `DECR` require a visit count of 4 before the node can be fired.

A statement which appears in the input source code may exhibit as a number of expression trees in the intermediate file. Except for one instance, the interpreter will attempt to finish traversing the whole expression tree before it relinquishes control back to the event scheduler. The only exception to this is the non-system function call. System function calls are assumed to be non-interruptible. This simply means that the interpreter will not relinquish control until a system function call is done regardless of how much time it might take the system function to execute. The process of traversing an expression tree is regarded by the simulator as an event.

3.4. The Task Control Block

In this section, we discuss the information necessary to save the status of a task. This information is kept in a data block named `TCB` (Task Control Block) defined in the file "`h.sim`". One `TCB` is associated with each task in the system. The items in a `TCB` which are related to the interpreter are:

1. Function Map: `subrtab[]`, is a table of all the function names defined in the input source code of a task.

2. Expression trees: `pgm[]` is a list of expression trees of a task.

3. Labels: `labtab[]` is a list of labels used in a task.

4. Data Memory: `mem_map[]` is the memory for variables and constants.

5. Register set: `R` is a pointer to register `[0]` of the register set.

6. Task and Processor: `tid` and `rid` are the task id and the processor id for the task, respectively. A task with `tid` equal to zero is the highest priority task, similarly, a processor with `rid` equal to zero is the master processor.

Creation and annihilation of tasks are described in the next section.

4. The Event Scheduler

The event scheduler is targeted largely for the SANDAC multiprocessor. Nevertheless, the event scheduler should be adaptable for other multiple processor architectures. They exist as a set of files under the directory "`../ran/`". It is assumed that the SANDAC multiprocessor has a global bus, a global multiplier and a global memory.

Other than those which are used to parameterize the expression trees, a number of parameters are used to parameterize the SANDAC multiprocessor at the system level. For instance, the variable `T_BUS` represents the time a processor takes to access the global bus if there is no conflict. Similarly, the variable `T_MEM` represents the time a processor takes to access the global memory provided there is no conflict. These variables are defined in the file "`h.vrtx`".

Events are ordered in a heap according to time of commencement and priority. Each processor can carry a maximum of `255` local tasks, where only one of them will be activated at one time. Each processor can maintain a maximum of `255` message queues created with the function `sc_rqcreate()` by any local tasks. Each queue can hold a maximum of `25` message pointers. These parameters are set in the file "`h.sim`" as `MAX_TASK`, `MAX_QID`, and `Q_SIZE`, respectively.

We use a heap data structure to order the events. Events are listed in the heap based on their execution times and priorities. The functions necessary for maintaining the heap data structure are `insert_heap()`, `delete_heap()` and `next()`. `Insert_heap()` inserts an eve into the heap, `delete_heap()` cancels an event that is already scheduled, `next()` retrieves the next event available for running, and `heapify()` restructure a non-heap structure into a heap.

The simulator is event-driven, which means that every action such as, interrupt, timeout, waiting, grasping resources, releasing resources, and execution is an event. Sometimes, a time-out event scheduled by the VRTX [1] call '`sc_qpend()`' has to be cancelled due to the arrival of a message before the time-out expires.

4.1. The Header Files

Five header files are directly related to the event scheduler. They are,

1. "h.heap": defines data structure for the heap-organized event list.
2. "h.vrtx": defines states and execution time of SANDAC related attributes.
3. "h.err": defines error codes for the VRTX primitives.
4. "h.err.msg": defines simulator error messages.
5. "h.para": defines timing of the intermediate nodes.

4.2. The Machine Dependent Functions

The files which contain machine dependent functions of the event scheduler are:

1. sys.c: this file contains the equivalence of some of the C library i/o and mathematical function calls and four VRTX primitives.
2. queue.c: this file contains functions which maintain (read, write) VRTX message queues and suspended task queues. For instance, enque() pushes a message pointer into a queue, and deque() retrieves a message pointer from a queue.
3. exec.c: this file contains a number of functions to mimic the interprocess communication protocols and multitasking on the SANDAC multiprocessor. The interprocess communication protocols are described [4]. Among those functions in this file, run() is responsible for directing the state transition of the tasks (see section 4.3). As the name suggests, Resume(), is for resuming the execution of an interrupted task. Scan_task() brings a task which is in a READY state back to the running state. Bootup() starts up a new task. Whoiswait() is evoked every time a message is deposited into a queue, it

checks to see whether there are any tasks pending for messages. `Resume_q()` resumes the execution of the highest priority task pending for the arrival of a message.

4. `main.c`: this file contains all the heap management functions such as `insert_heap()`, `next()`, etc. The routine `stat()` in this file gathers statistics at each evocation of event.

5. `reader.c`: this file contains a function `reader()` which is responsible for reading the intermediate files (tasks). Each task has it own task control block (TCB). The relationship among tasks and processors is maintained by a data structure `RID` which is defined in the header file "`h.sim`". Reading an intermediate file is the reversal of what the parser does. `Eread()` reconstructs the parse trees, bookkeeping the functions and labels, `rdfloat()` assigns storage locations for floating-point constants, `rdin()` deals with integer constants.

`Reader()` can read multiple tasks. The assignment of tasks to processors is according to the following simple rule. Let $m \geq 1$ denotes the number of processors that a user acquires; let $task[j]$ $\{j=1,..,n\}$ be the list of tasks that a user supplied from the command line. $Task[i]$ is assigned to processor $(i \ mod \ m)+1$ with priority i/m.

4.2.1. The VRTX C Library

In SANDAC, tasks communicate with one another by sending and receiving pointer-sized, nonzero messages via VRTX-control structures known as mailboxes and queues. The current installation of the simulation includes only four VRTX primitives which maintain message queues. They are,

1. `sc_rqcreate(rid,qid,size,&err)`: creates a queue.

2. `sc_rqpost(rid,qid,&msg,&err)`: posts message to a queue.

3. `sc_rqaccept(rid,qid,&err)`: reads a message from a queue.

4. `sc_qpend(qid,timeout,&err)`: pends for a message at a queue.

where `rid` is the remote processor id; `qid` is the queue id; `size` is the maximum number of messages that a queue can hold; `msg` is a pointer to a message; `timeout` is a time-out count; and `err` is a variable for holding the completion return code.

4.2.2. The Standard C Library

Some of the functions in the standard C Library for i/o and string operations have also been implemented. Namely,

1. `getchar(stream)`: read a character from standard input.

2. `printf(format, [,arg])`: print out to standard output.

3. `strcpy(d_string, s_string)`: string copy.

4. `times(buffer)`: get process times.

5. `strcmp(string_1, string_2)`: string compare.

The detailed description of these functions can be found in the standard C library.

4.2.3. The Standard C Mathematical Library

All of the mathematical functions appearing in the standard C Math library are included in the current installation. Among those functions are: `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, `exp()`, `log()`, `log10()`, `pow()`, and `sqrt()`. The description of these functions can be found in the standard C Math library.

4.3. Task States and State Transitions

Sandy has provisions for simulating multitasking and parallel processing. More than one task can be allocated to each processor, and there are multiple processors in SANDAC. The state of a processor is defined to be the state of its active task. A task is active if it is not in the READY or SLEEP state. A tasks would be in one of the following states:

1. **RUN**: the task has control of the processor and is executing its assigned instruction stream.

2. **PEND**: the task suspends in mid-execution and is waiting to be readied by a system call or an event, such as waiting for a message to arrive.

3. **SWI**: the task is software interrupted by another processor which requires immediate service. The execution of the task is temporarily suspended and would be resume as soon as the request is serviced.

4. **READY**: the task is ready for execution but cannot gain control of the processor until all higher priority tasks existing in the ready or execution state are either completed or suspended.

5. **PROBE**: the task is trying to interrupt another task which is not ready to be interrupted.

6. **MAIL**: the task is resumed from pending due to the arrival of a message.

7. **WAKEUP**: the task is awakened from suspension due to time-out.

8. **BAC**: the task is trying to grasp the global bus.

9. **BAJ**: the task couldn't get hold of the global bus due to a possible conflict.

10. **SLEEP**: the task is counting down the timeout.

11. **TAS**: the task is trying to get exclusive access of the global control block.

12. **ATCB**: the task is accessing the global communication block. le **DEAD**: the task exhausted its instruction stream and ended.

A processor can only be interrupted if it is either in the RUN or SLEEP state. For instance, when a `sc_rqaccept()` function is encountered in the instruction stream of a task, the task may go through a number of states before the message can be deposited into the remote queue. First, this task exits from its RUN state and enters into the BUS state where the (local) processor will try to see whether there is any other processor which is holding the bus. If the bus is free, then the processor will try to get exclusive access to the remote processor's global communication block by entering into the TAS state. After that, the local processor, in the SWI state, issues a software interrupt to the remote processor indicating that it has a request which it must process. The local processor will then halt by putting itself into the PEND state. If condition is favorable, the remote processor (in the ATCB state) accepts this software interrupt by issuing a local read queue function to deposit the message into the appropriate queue. The remote processor would then issue an interrupt to the local processor to wake it back up. The local and remote processors may now resume their execution in the RUN state.

The current, previous state, and next state of a task are defined respectively by the variables `c_state`, `p_state`, and `n_state`, in the task control block (TCB). The variable `resource` indicates what resource the task is currently holding.

4.4. The Global Processor Control Block

A processor identifier (`rid`) and a global processor control block (RID) are assigned to each processor. These structures, which are used for inter-processor communication and control, will be referenced by the remote processor identifier assigned to each processor and contain the processor's current status and its communication table. It is assumed that the global processor control block is kept in the global memory. Each global processor control block contains,

1. `Pid[]`: a list of task control block for all the (local) tasks allocated.

2. `Tasks`: the number of task allocated.

3. `Active_pid`: a pointer to the task control block of the active task.

4. `Qid[]`: a list of all the `qid` created by the local tasks.

5. `Queue[]`: a list of all the queues.

6. `Req_rid`, `req_qid`, `req_code`, and `q_msg`: are used in remote queueing.

5. Statistics Collection

Statistics can be collected in the function `stat()`. A Gantt chart can also be produced when evoking "`sandy`" with the "`s`" option. For instance,

```
sandy -2 -s task11 task12
```

will cause the simulator to generate a Gantt chart into the file named "`g.chart`" in the current directory. The current states of the processors are reported every `INC_CYC` number of machine cycles. The parameter `INC_CYC` is defined in the file "`main.c`".

6. User Supplied Timing Parameters

The interpreter relies on the user to supply a set of timing parameters defining the execution time of each intermediate node. There are 80 different type of nodes and 5 data types. So, all together they provide a maximum of 400 different contexts by which one could assign timing to the simulator. This parameters are defined in the file "`h.para`".

Calibrating these parameters is a compiler related issue. It requires one's expertise in correlating these intermediate nodes with the assembly code that would be generated by the compiler which actually produces the code for the machine. Preliminary conversation with Mr. Carl Rosenberg of GREENHILL Compiler Company indicates

that such a correspondence exists.

7. Options for Debugging

We don't expect the simulator to be perfect, despite the fact that efforts have been made to test most of the source code. The interpreter in particular has been tested quite extensively using 10 benchmarks. Debugging codes can be generated when compiling the source code with the appropriate options such as `DEBUG1` `DEBUG2`, and `DEBUG3`. These options can be set or unset in the "`Makefile`"s.

Token	Timing	Meaning
T_SWITCH	(double) 1000	/* switch statement */
T_COTXT_SW	(double) 2000	/* context switch */
T_JMP	(double) 2000	/* branch or jump */
T_RET	(double) 2000	/* return from function */
T_RVAL	(double) 2000	/* force to return register */
T_NAME	(double) 2000	/* direct fetch a variable */
T_ICON	(double) 2000	/* fetch an immediate constant */
T_REG_I	(double) 2000	/* fetch register integer */
T_REG_F	(double) 2000	/* fetch register float */
T_REG_D	(double) 2000	/* fetch register double */
T_RSUB	(double) 2000	/* register subtract */
T_TOSTK	(double) 2000	/* from register to stack */
T_GESTACK	(double) 2000	/* acquiring stack space */
T_RMSTACK	(double) 2000	/* relinquish stack space */
T_UMUL	(double) 2000	/* indirect fetch for data */
T_FORCE	(double) 2000	/* force value to a register */
T_NOT	(double) 2000	/* ! operator */
T_UMINUS_I	(double) 2000	/* indirect get integer */
T_UMINUS_F	(double) 2000	/* indirect get float */
T_UMINUS_D	(double) 2000	/* indirect get double */
T_SCONV_IF	(double) 2000	/* shape conversion to integer */
T_SCONV_II	(double) 2000	/* shape conversion to integer */
T_SCONV_ID	(double) 2000	/* shape conversion to integer */
T_SCONV_FI	(double) 2000	/* shape conversion to float */
T_SCONV_FD	(double) 2000	/* shape conversion to float */
T_SCONV_DI	(double) 2000	/* shape conversion to double */
T_SCONV_DF	(double) 2000	/* shape conversion to double */
T_ANDAND	(double) 2000	/* && operator */
T_OROR	(double) 2000	/* || operator */
T_RISCOR	(double) 2000	/* Xor operator */
T_PLUS_I	(double) 2000	/* integer plus */
T_PLUS_F	(double) 2000	/* float plus */
T_PLUS_D	(double) 2000	/* double plus */
T_MINUS_I	(double) 2000	/* integer minus */
T_MINUS_F	(double) 2000	/* float minus */
T_MINUS_D	(double) 2000	/* double minus */
T_MUL_I	(double) 2000	/* integer multiply */
T_MUL_D	(double) 2000	/* float multiply */
T_MUL_F	(double) 2000	/* double multiply */
T_DIV_I	(double) 2000	/* integer divide */
T_DIV_F	(double) 2000	/* float divide */
T_DIV_D	(double) 2000	/* double divide */
T_MOD_I	(double) 2000	/* % operator */
T_AND_I	(double) 2000	/* and operator */
T_OR_I	(double) 2000	/* or operator */
T_ER_I	(double) 2000	/* xor operator */
T_LS_I	(double) 2000	/* integer left shift */
T_RS_I	(double) 2000	/* integer right shift */
T_GT_I	(double) 2000	/* greater than integer */
T_GT_D	(double) 2000	/* greater than double */
T_GT_F	(double) 2000	/* greater than float */
T_GE_I	(double) 2000	/* greater than or equal integer */
T_GE_D	(double) 2000	/* greater than or equal double */
T_GE_F	(double) 2000	/* greater than or equal float */
T_LE_I	(double) 2000	/* less than or equal integer */
T_LE_D	(double) 2000	/* less than or equal to double */
T_LE_F	(double) 2000	/* less than or equal to float */
T_LT_I	(double) 2000	/* less than integer */
T_LT_D	(double) 2000	/* less than double */
T_LT_F	(double) 2000	/* less than float */
T_EQ_I	(double) 2000	/* equal to integer */
T_EQ_F	(double) 2000	/* equal to float */
T_EQ_D	(double) 2000	/* equal to double */
T_NE_I	(double) 2000	/* not equal to integer */
T_NE_D	(double) 2000	/* not equal to double */
T_NE_F	(double) 2000	/* not equal to float */
T_ASSIGN_REG	(double) 2000	/* assignment to register */
T_ASSIGN_NAME	(double) 2000	/* assignment to memory */
T_UMUL_ASS	(double) 2000	/* assignment indirect */
T_STASG	(double) 2000	/* assignment structure */
T_SYS_CALL	(double) 2000	/* system calls */
T_TA_SWITCH	(double) 2000	/* overhead in task switching */

Table 2. Tentative Timing Parameters

8. Installation

The parser has been developed on UNIX 4.2. The simulator has been developed on UTX/32 1.1/C. We tried to avoid writting installation dependent code as much as possible. All of the system calls used in the source code should be available from "the standard I/O library". .

9. Remarks

Appendix A presents the source code and its parse trees of one of the widely respected C benchmarks - `whetstone.c`, which `sandy` has been tested on. Appendix B presents the source codes of a set of tested programs that we used to demonstrate the capability of `sandy` to simulate the multitasking and message passing environments on SANDAC.

10. Acknowledgments

Many thanks to Miquel Huguet for his generous help with the Portable C Compiler. We are also thankful to George Davidson of Sandia National Laboratories for his patience and support throughout this project.

REFERENCES

[1] *VRTX C Interface Library User's Guide*: Hunter & Ready, Inc., November 1984.

[2] C. R. Borgman and P. E. Pierce, "A Hardware/Software System for Advanced Development Guidance and Control Experiments," in *AIAA Computers in Aerospace Conference*, Hartford CT: Oct. 1983, pp. 377-384.

[3] William J. Garrison, *CACI NETWORK II.5*, Los Angeles, California: CACI, Inc.-Federal, April 1984.

[4] David L. Harris, *Inter-Processor Communication Introduction*: Sandia National Laboratory Internal Report, May 3, 1984.

[5] Steve C. Johnson, *A Tour Through the Portable C Compiler*: Bell Laboratories, 1975.

[6] Brain W. Kernighan and Dennis M. Ritchie, *The C Programming Language*: Prentice-Hall Series, 1978.

[7] John Lions, *The Second Pass of the Portable C Compiler*: Bell Laboratories, June 1979.

Appendix A

```
/*
 *      Whetstone benchmark in C.  This program is a translation of the
 *      original Algol version in "A Synthetic Benchmark" by H.J. Curnow
 *      and B.A. Wichman in Computer Journal, Vol  19 #1, February 1976.
 *
 *      Used to test floating point and library functions performance
 *      in SINGLE processor case.
 *
 *      Generate parse tree by :  cc -E -DPOUT whet.c | pas > parse
 *      run simulation by      :  sandy -1 parse
 */

#define ITERATIONS      2 /* set number of Whetstone instructions */

#include <math.h>

double          x1, x2, x3, x4, x, y, z, t, t1, t2;
double          e1[4];
int             i, j, k, l, n1, n2, n3, n4, n6, n7, n8, n9, n10, n11;

main()
{

        /* initialize constants */

        t   =   0.499975;
        t1  =   0.50025;
        t2  =   2.0;

        /* set values of module weights */

        n1  =   0 * ITERATIONS;
        n2  =  12 * ITERATIONS;
        n3  =  14 * ITERATIONS;
        n4  = 345 * ITERATIONS;
        n6  = 210 * ITERATIONS;
        n7  =  32 * ITERATIONS;
        n8  = 899 * ITERATIONS;
        n9  = 616 * ITERATIONS;
        n10 =   0 * ITERATIONS;
        n11 =  93 * ITERATIONS;

/* MODULE 1:  simple identifiers */

        x1 =  1.0;
        x2 = x3 = x4 = -1.0;

        for(i = 1; i <= n1; i += 1) {
                x1 = ( x1 + x2 + x3 - x4 ) * t;
                x2 = ( x1 + x2 - x3 - x4 ) * t;
                x3 = ( x1 - x2 + x3 + x4 ) * t;
                x4 = (-x1 + x2 + x3 + x4 ) * t;
        }
#ifdef POUT
        pout(n1, n1, n1, x1, x2, x3, x4);
#endif


/* MODULE 2:  array elements */

        e1[0] =  1.0;
        e1[1] = e1[2] = e1[3] = -1.0;

        for (i = 1; i <= n2; i +=1) {
```

```
                e1[0] = ( e1[0] + e1[1] + e1[2] - e1[3] ) * t;
                e1[1] = ( e1[0] + e1[1] - e1[2] + e1[3] ) * t;
                e1[2] = ( e1[0] - e1[1] + e1[2] + e1[3] ) * t;
                e1[3] = (-e1[0] + e1[1] + e1[2] + e1[3] ) * t;
        }
#ifdef POUT
        pout(n2, n3, n2, e1[0], e1[1], e1[2], e1[3]);
#endif

/* MODULE 3:  array as parameter */

        for (i = 1; i <= n3; i += 1)
                pa(e1);
#ifdef POUT
        pout(n3, n2, n2, e1[0], e1[1], e1[2], e1[3]);
#endif

/* MODULE 4:  conditional jumps */

        j = 1;
        for (i = 1; i <= n4; i += 1) {
                if (j == 1)
                        j = 2;
                else
                        j = 3;

                if (j > 2)
                        j = 0;
                else
                        j = 1;

                if (j < 1 )
                        j = 1;
                else
                        j = 0;
        }
#ifdef POUT
        pout(n4, j, j, x1, x2, x3, x4);
#endif

/* MODULE 5:  omitted */

/* MODULE 6:  integer arithmetic */

        j = 1;
        k = 2;
        l = 3;

        for (i = 1; i <= n6; i += 1) {
                j = j * (k - j) * (l -k);
                k = l * k - (l - j) * k;
                l = (l - k) * (k + j);

                e1[l - 2] = j + k + l;          /* C arrays are zero based */
                e1[k - 2] = j * k * l;
        }
#ifdef POUT
        pout(n6, j, k, e1[0], e1[1], e1[2], e1[3]);
#endif

/* MODULE 7:  trig. functions */

        x = y = 0.5;
```

```
        for(i = 1; i <= n7; i +=1) {
                x = t * atan(t2*sin(x)*cos(x)/(cos(x+y)+cos(x-y)-1.0));
                y = t * atan(t2*sin(y)*cos(y)/(cos(x+y)+cos(x-y)-1.0));
        }
#ifdef POUT
        pout(n7, j, k, x, x, y, y);
#endif

/* MODULE 8:  procedure calls */

        x = y = z = 1.0;

        for (i = 1; i <= n8; i +=1)
                p3(x, y, &z);
#ifdef POUT
        pout(n8, j, k, x, y, z, z);
#endif

/* MODULE9:  array references */

        j = 1;
        k = 2;
        l = 3;

        e1[0] = 1.0;
        e1[1] = 2.0;
        e1[2] = 3.0;

        for(i = 1; i <= n9; i += 1)
                p0();
#ifdef POUT
        pout(n9, j, k, e1[0], e1[1], e1[2], e1[3]);
#endif

/* MODULE10:  integer arithmetic */

        j = 2;
        k = 3;

        for(i = 1; i <= n10; i +=1) {
                j = j + k;
                k = j + k;
                j = k - j;
                k = k - j - j;
        }
#ifdef POUT
        pout(n10, j, k, x1, x2, x3, x4);
#endif

/* MODULE11:  standard functions */

        x = 0.75;
        for(i = 1; i <= n11; i +=1)
                x = sqrt( exp( log(x) / t1));

#ifdef POUT
        pout(n11, j, k, x, x, x, x);
#endif
}

pa(e)
double e[4];
{
        register int j;
```

```
        j = 0;
    lab:
        e[0] = (  e[0] + e[1] + e[2] - e[3] ) * t;
        e[1] = (  e[0] + e[1] - e[2] + e[3] ) * t;
        e[2] = (  e[0] - e[1] + e[2] + e[3] ) * t;
        e[3] = ( -e[0] + e[1] + e[2] + e[3] ) / t2;
        j += 1;
        if (j < 6)
                goto lab;
}


p3(x, y, z)
double x, y, *z;
{
        x  = t * (x + y);
        y  = t * (x + y);
        *z = (x + y) /t2;
}


p0()
{
        e1[j] = e1[k];
        e1[k] = e1[l];
        e1[l] = e1[j];
}

#ifdef POUT
pout(n, j, k, x1, x2, x3, x4)
int n, j, k;
double x1, x2, x3, x4;
{
        printf("%6d%6d%6d  %5e  %5e  %5e  %5e\n",
                n, j, k, x1, x2, x3, x4);
}
#endif
```

```
LL0:
        .data
        .comm   _x1,1
        .comm   _x2,1
        .comm   _x3,1
        .comm   _x4,1
        .comm   _x,1
        .comm   _y,1
        .comm   _z,1
        .comm   _t,1
        .comm   _t1,1
        .comm   _t2,1
        .comm   _e1,4
        .comm   _i,1
        .comm   _j,1
        .comm   _k,1
        .comm   _l,1
        .comm   _n1,1
        .comm   _n2,1
        .comm   _n3,1
        .comm   _n4,1
        .comm   _n6,1
        .comm   _n7,1
        .comm   _n8,1
        .comm   _n9,1
        .comm   _n10,1
        .comm   _n11,1
        .text
        .globl  _main
)_main:
.0
124     1       1       4       L76
[74     0       30
        .data
L77:
        .double 4.99974966049194340000e-01
        .text
.25     "whet.c"
58      7
2       0       0       7       _t
2       0       0       7       L77
        .data
L78:
        .double 5.00249981880187990000e-01
        .text
.26     "whet.c"
58      7
2       0       0       7       _t1
2       0       0       7       L78
        .data
L79:
        .double 2.00000000000000000000e+00
        .text
.27     "whet.c"
58      7
2       0       0       7       _t2
2       0       0       7       L79
.31     "whet.c"
58      4
2       0       0       4       _n1
4       0       0       4
.32     "whet.c"
58      4
2       0       0       4       _n2
```

```
4       24      0       4
.33     "whet.c"
58      4
2       0       0       4       _n3
4       28      0       4
.34     "whet.c"
58      4
2       0       0       4       _n4
4       690     0       4
.35     "whet.c"
58      4
2       0       0       4       _n6
4       420     0       4
.36     "whet.c"
58      4
2       0       0       4       _n7
4       64      0       4
.37     "whet.c"
58      4
2       0       0       4       _n8
4       1798    0       4
.38     "whet.c"
58      4
2       0       0       4       _n9
4       1232    0       4
.39     "whet.c"
58      4
2       0       0       4       _n10
4       0       0       4
.40     "whet.c"
58      4
2       0       0       4       _n11
4       186     0       4
        .data
L80:
        .double 1.00000000000000000000e+00
        .text
.44     "whet.c"
58      7
2       0       0       7       _x1
2       0       0       7       L80
        .data
L81:
        .double -1.00000000000000000000e+00
        .text
.45     "whet.c"
58      7
2       0       0       7       _x2
58      7
2       0       0       7       _x3
58      7
2       0       0       7       _x4
2       0       0       7       L81
.47     "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L82
)       nop
L83:
[74     0       30
.48     "whet.c"
58      7
```

```
2       0       0       7       _x1
11      7
8       7
6       7
6       7
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
2       0       0       7       _t
.49     "whet.c"
58      7
2       0       0       7       _x2
11      7
8       7
8       7
6       7
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
2       0       0       7       _t
.50     "whet.c"
58      7
2       0       0       7       _x3
11      7
6       7
6       7
8       7
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
2       0       0       7       _t
.51     "whet.c"
58      7
2       0       0       7       _x4
11      7
6       7
6       7
6       7
10      0       7
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
2       0       0       7       _t
L84:
.52     "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L82:
.52     "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n1
4       83      0       4
L85:
.54     "whet.c"
70      4
4       0       0       224     _pout
```

```
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n1
2       0       0       4       _n1
2       0       0       4       _n1
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
        .data
L87:
        .double 1.00000000000000000000e+00
        .text
.60     "whet.c"
58      7
2       0       0       7       _e1
2       0       0       7       L87
        .data
L88:
        .double -1.00000000000000000000e+00
        .text
.61     "whet.c"
58      7
2       1       0       7       _e1
58      7
2       2       0       7       _e1
58      7
2       3       0       7       _e1
2       0       0       7       L88
.63     "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L89
)       nop
L90:
[74     0       30
.64     "whet.c"
58      7
2       0       0       7       _e1
11      7
8       7
6       7
6       7
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
2       0       0       7       _t
.65     "whet.c"
58      7
2       1       0       7       _e1
11      7
6       7
8       7
6       7
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
```

```
2       3       0       7       _e1
2       0       0       7       _t
.66     "whet.c"
58      7
2       2       0       7       _e1
11      7
6       7
6       7
8       7
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
2       0       0       7       _t
.67     "whet.c"
58      7
2       3       0       7       _e1
11      7
6       7
6       7
6       7
10      0       7
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
2       0       0       7       _t
L91:
.68     "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L89:
.68     "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n2
4       90      0       4
L92:
.70     "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n2
2       0       0       4       _n3
2       0       0       4       _n2
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
.75     "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L93
)       nop
```

```
L94:
.76     "whet.c"
70      4
4       0       0       224     _pa
4       0       0       27      _e1
L95:
.76     "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L93:
.76     "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n3
4       94      0       4
L96:
.78     "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n3
2       0       0       4       _n2
2       0       0       4       _n2
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
.83     "whet.c"
58      4
2       0       0       4       _j
4       1       0       4
.84     "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L98
)       nop
L99:
[74     0       30
.85     "whet.c"
109     4
80      4
2       0       0       4       _j
4       1       0       4
4       102     0       4
.86     "whet.c"
58      4
2       0       0       4       _j
4       2       0       4
.0
114     0       0       4       L103
)       nop
L102:
.88     "whet.c"
58      4
```

```
2       0       0       4       _j
4       3       0       4
L103:
.90     "whet.c"
109     4
85      4
2       0       0       4       _j
4       2       0       4
4       104     0       4
.91     "whet.c"
58      4
2       0       0       4       _j
4       0       0       4
.0
114     0       0       4       L105
)       nop
L104:
.93     "whet.c"
58      4
2       0       0       4       _j
4       1       0       4
L105:
.95     "whet.c"
109     4
83      4
2       0       0       4       _j
4       1       0       4
4       106     0       4
.96     "whet.c"
58      4
2       0       0       4       _j
4       1       0       4
.0
114     0       0       4       L107
)       nop
L106:
.98     "whet.c"
58      4
2       0       0       4       _j
4       0       0       4
L107:
L100:
.99     "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L98:
.99     "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n4
4       99      0       4
L101:
.101    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
```

```
2       0       0       4       _n4
2       0       0       4       _j
2       0       0       4       _j
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
.108    "whet.c"
58      4
2       0       0       4       _j
4       1       0       4
.109    "whet.c"
58      4
2       0       0       4       _k
4       2       0       4
.110    "whet.c"
58      4
2       0       0       4       _l
4       3       0       4
.112    "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L108
)       nop
L109:
[74     0       30
.113    "whet.c"
58      4
2       0       0       4       _j
11      4
11      4
2       0       0       4       _j
8       4
2       0       0       4       _k
2       0       0       4       _j
8       4
2       0       0       4       _l
2       0       0       4       _k
.114    "whet.c"
58      4
2       0       0       4       _k
8       4
11      4
2       0       0       4       _l
2       0       0       4       _k
11      4
8       4
2       0       0       4       _l
2       0       0       4       _j
2       0       0       4       _k
.115    "whet.c"
58      4
2       0       0       4       _l
11      4
8       4
2       0       0       4       _l
2       0       0       4       _k
6       4
2       0       0       4       _k
2       0       0       4       _j
.117    "whet.c"
58      7
```

```
13      0       7
6       27
2       0       0       4       _l
4       -2      0       27      _e1
104     0       7
6       4
6       4
2       0       0       4       _j
2       0       0       4       _k
2       0       0       4       _l
.118    "whet.c"
58      7
13      0       7
6       27
2       0       0       4       _k
4       -2      0       27      _e1
104     0       7
11      4
11      4
2       0       0       4       _j
2       0       0       4       _k
2       0       0       4       _l
L110:
.119    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L108:
.119    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n6
4       109     0       4
L111:
.121    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n6
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
        .data
L112:
        .double 5.00000000000000000000e-01
        .text
.126    "whet.c"
58      7
2       0       0       7       _x
58      7
2       0       0       7       _y
2       0       0       7       L112
.128    "whet.c"
58      4
```

```
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L113
)       nop
L114:
[74     0       30
        .data
L117:
        .double 1.00000000000000000000e+00
        .text
.129    "whet.c"
58      7
2       0       0       7       _x
11      7
2       0       0       7       _t
70      7
4       0       0       227     _atan
60      7
11      7
11      7
2       0       0       7       _t2
70      7
4       0       0       227     _sin
2       0       0       7       _x
70      7
4       0       0       227     _cos
2       0       0       7       _x
8       7
6       7
70      7
4       0       0       227     _cos
6       7
2       0       0       7       _x
2       0       0       7       _y
70      7
4       0       0       227     _cos
8       7
2       0       0       7       _x
2       0       0       7       _y
2       0       0       7       L117
        .data
L118:
        .double 1.00000000000000000000e+00
        .text
.130    "whet.c"
58      7
2       0       0       7       _y
11      7
2       0       0       7       _t
70      7
4       0       0       227     _atan
60      7
11      7
11      7
2       0       0       7       _t2
70      7
4       0       0       227     _sin
2       0       0       7       _y
70      7
4       0       0       227     _cos
2       0       0       7       _y
8       7
6       7
```

```
70      7
4       0       0       227     _cos
6       7
2       0       0       7       _x
2       0       0       7       _y
70      7
4       0       0       227     _cos
8       7
2       0       0       7       _x
2       0       0       7       _y
2       0       0       7       L118
L115:
.131    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L113:
.131    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n7
4       114     0       4
L116:
.133    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n7
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _x
2       0       0       7       _x
2       0       0       7       _y
2       0       0       7       _y
        .data
L119:
        .double 1.0000000000000000000e+00
        .text
.138    "whet.c"
58      7
2       0       0       7       _x
58      7
2       0       0       7       _y
58      7
2       0       0       7       _z
2       0       0       7       L119
.140    "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L120
)       nop
L121:
.141    "whet.c"
70      4
4       0       0       224     _p3
```

```
56      4
56      4
2       0       0       7       _x
2       0       0       7       _y
4       0       0       27      _z
L122:
.141    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L120:
.141    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n8
4       121     0       4
L123:
.143    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n8
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _x
2       0       0       7       _y
2       0       0       7       _z
2       0       0       7       _z
.148    "whet.c"
58      4
2       0       0       4       _j
4       1       0       4
.149    "whet.c"
58      4
2       0       0       4       _k
4       2       0       4
.150    "whet.c"
58      4
2       0       0       4       _l
4       3       0       4
        .data
L125:
        .double 1.00000000000000000000e+00
        .text
.152    "whet.c"
58      7
2       0       0       7       _e1
2       0       0       7       L125
        .data
L126:
        .double 2.00000000000000000000e+00
        .text
.153    "whet.c"
58      7
2       1       0       7       _e1
2       0       0       7       L126
        .data
```

```
L127:
        .double 3.00000000000000000000e+00
        .text
.154    "whet.c"
58      7
2       2       0       7       _e1
2       0       0       7       L127
.156    "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L128
)       nop
L129:
.157    "whet.c"
72      0       4
4       0       0       224     _p0
L130:
.157    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L128:
.157    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n9
4       129     0       4
L131:
.159    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n9
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _e1
2       1       0       7       _e1
2       2       0       7       _e1
2       3       0       7       _e1
.164    "whet.c"
58      4
2       0       0       4       _j
4       2       0       4
.165    "whet.c"
58      4
2       0       0       4       _k
4       3       0       4
.167    "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L133
)       nop
L134:
```

```
[74     0       30
.168    "whet.c"
58      4
2       0       0       4       _j
6       4
2       0       0       4       _j
2       0       0       4       _k
.169    "whet.c"
58      4
2       0       0       4       _k
6       4
2       0       0       4       _j
2       0       0       4       _k
.170    "whet.c"
58      4
2       0       0       4       _j
8       4
2       0       0       4       _k
2       0       0       4       _j
.171    "whet.c"
58      4
2       0       0       4       _k
8       4
8       4
2       0       0       4       _k
2       0       0       4       _j
2       0       0       4       _j
L135:
.172    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L133:
.172    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n10
4       134     0       4
L136:
.174    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n10
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _x1
2       0       0       7       _x2
2       0       0       7       _x3
2       0       0       7       _x4
        .data
L137:
        .double 7.50000000000000000000e-01
        .text
.179    "whet.c"
58      7
2       0       0       7       _x
```

```
2       0       0       7       L137
.180    "whet.c"
58      4
2       0       0       4       _i
4       1       0       4
.0
114     0       0       4       L138
)       nop
L139:
.181    "whet.c"
58      7
2       0       0       7       _x
70      7
4       0       0       227     _sqrt
70      7
4       0       0       227     _exp
60      7
70      7
4       0       0       227     _log
2       0       0       7       _x
2       0       0       7       _t1
L140:
.181    "whet.c"
7       4
2       0       0       4       _i
4       1       0       4
L138:
.181    "whet.c"
109     4
76      0       4
82      4
2       0       0       4       _i
2       0       0       4       _n11
4       139     0       4
L141:
.184    "whet.c"
70      4
4       0       0       224     _pout
56      4
56      4
56      4
56      4
56      4
56      4
2       0       0       4       _n11
2       0       0       4       _j
2       0       0       4       _k
2       0       0       7       _x
2       0       0       7       _x
2       0       0       7       _x
2       0       0       7       _x
L75:
.0
125     1       1       4       L76
.0
115     0       0       4
)       nop
]76     0
        .data
        .text
        .globl  _pa
)_pa:
.0
124     1       1       4       L144
```

```
[142    0       28
.193    "whet.c"
58      4
94      0       29      4
4       0       0       4
L145:
.195    "whet.c"
58      7
13      0       7
6       27
94      0       30      27
4       0       0       4
11      7
8       7
6       7
6       7
13      0       7
6       27
94      0       30      27
4       0       0       4
13      0       7
6       27
94      0       30      27
4       1       0       4
13      0       7
6       27
94      0       30      27
4       2       0       4
13      0       7
6       27
94      0       30      27
4       3       0       4
2       0       0       7       _t
.196    "whet.c"
58      7
13      0       7
6       27
94      0       30      27
4       1       0       4
11      7
6       7
8       7
6       7
13      0       7
6       27
94      0       30      27
4       0       0       4
13      0       7
6       27
94      0       30      27
4       1       0       4
13      0       7
6       27
94      0       30      27
4       2       0       4
13      0       7
6       27
94      0       30      27
4       3       0       4
2       0       0       7       _t
.197    "whet.c"
58      7
13      0       7
6       27
```

```
94      0       30      27
4       2       0       4
11      7
6       7
6       7
8       7
13      0       7
6       27
94      0       30      27
4       0       0       4
13      0       7
6       27
94      0       30      27
4       1       0       4
13      0       7
6       27
94      0       30      27
4       2       0       4
13      0       7
6       27
94      0       30      27
4       3       0       4
2       0       0       7       _t
.198    "whet.c"
58      7
13      0       7
6       27
94      0       30      27
4       3       0       4
60      7
6       7
6       7
6       7
10      0       7
13      0       7
6       27
94      0       30      27
4       0       0       4
13      0       7
6       27
94      0       30      27
4       1       0       4
13      0       7
6       27
94      0       30      27
4       2       0       4
13      0       7
6       27
94      0       30      27
4       3       0       4
2       0       0       7       _t2
.199    "whet.c"
7       4
94      0       29      4
4       1       0       4
.200    "whet.c"
109     4
83      4
94      0       29      4
4       6       0       4
4       146     0       4
.0
114     0       0       4       L145
)       nop
```

```
L146:
L143:
.0
125     1       1       4       L144
.0
115     0       0       4
)       nop
]144    0
        .data
        .text
        .globl  _p3
)_p3:
.0
124     1       1       4       L149
[147    0       25
.208    "whet.c"
58      7
94      0       29      7
11      7
2       0       0       7       _t
6       7
94      0       29      7
94      0       27      7
.209    "whet.c"
58      7
94      0       27      7
11      7
2       0       0       7       _t
6       7
94      0       29      7
94      0       27      7
.210    "whet.c"
58      7
13      0       7
94      0       26      27
60      7
6       7
94      0       29      7
94      0       27      7
2       0       0       7       _t2
L148:
.0
125     1       1       4       L149
.0
115     0       0       4
)       nop
]149    0
        .data
        .text
        .globl  _p0
)_p0:
.0
124     1       1       4       L152
[150    0       30
.216    "whet.c"
58      7
13      0       7
6       27
2       0       0       4       _j
4       0       0       27      _e1
13      0       7
6       27
2       0       0       4       _k
4       0       0       27      _e1
```

```
.217    "whet.c"
58      7
13      0       7
6       27
2       0       0       4       _k
4       0       0       27      _e1
13      0       7
6       27
2       0       0       4       _l
4       0       0       27      _e1
.218    "whet.c"
58      7
13      0       7
6       27
2       0       0       4       _l
4       0       0       27      _e1
13      0       7
6       27
2       0       0       4       _j
4       0       0       27      _e1
L151:
.0
125     1       1       4       L152
.0
115     0       0       4
)       nop
]152    0
        .data
        .text
        .globl  _pout
)_pout:
.0
124     1       1       4       L155
[153    0       19
        .data   1
L157:
        .ascii  "%6d%6d%6d  %5e  %5e  %5e  %5e\n\0"
        .text
.227    "whet.c"
70      4
4       0       0       224     _printf
56      4
56      4
56      4
56      4
56      4
56      4
56      4
4       0       0       22      L157
94      0       30      4
94      0       29      4
94      0       28      4
94      0       26      7
94      0       24      7
94      0       22      7
94      0       20      7
L154:
.0
125     1       1       4       L155
.0
115     0       0       4
)       nop
]155    0
        .data
```

Appendix B

```
/* task 2  talk to task 1 */
char * sc_rqaccept();
main()
{
    int err;
    char *msg, *his_msg;

    printf("P2: task created\n");
    sc_rqcreate(0,0x11,10,&err);  /* create a local message queue */
    printf("P2: error code for create %d\n",err);
    printf("    >> P2: -- delaying 1.\n");
    printf("    >> P2: -- delaying 2.\n");
    printf("    >> P2: -- delaying 3.\n");
    printf("    >> P2: -- delaying 4.\n");
    printf("    >> P2: -- delaying 5.\n");
    printf("    >> P2: -- delaying 6.\n");
    printf("    >> P2: -- delaying 7.\n");
    printf("    >> P2: -- delaying 8.\n");
    printf("    >> P2: -- delaying 9.\n");

    his_msg=sc_rqaccept(0,0x11,&err);  /* accepting a message for queue 0x11*/

    printf("P2: error %d\n",err);
    printf("P2: error %d\n",err);
    printf("P2: error %d\n",err);
    printf("P2: error %d\n",err);
    printf("P2: error %d\n",err);
    printf("P2: error %d\n",err);
    if(err == 0) printf("P2: message received is %s\n",his_msg);
    else printf("P2: error in message accept ..\n");

    printf("P2: error code for accept %d\n",err);
    printf("P2: END of task 2\n");
}
```

```
/* demo
*  try:
*       cc -E task1.c | pas > tree1
*       cc -E task2.c | pas > tree2
*       cc -E task4.c | pas > tree4
*
*  to generate the parse trees.
*
*  then run simulation by saying
*
*       sandy -1 tree*                          ; one processor 3 tasks
*  or
*       sandy -2 tree*                          ; two processors 3 tasks
*  or
*       sandy -3 tree*                          ; three processors 3 tasks
*  etc.
*/
#include <math.h>
char * sc_rqaccept();
char * sc_qpend();
main()              /* main task for processor 1 */
{
    int    err;
    char   *my_msg;
    double timeout;
    double test;

    printf("P1: task created\n");
    test=sin(1.2);
    printf("P1: Sin value is %e\n",test);

    timeout=0.0;     /* wait for ever for message  */
    sc_rqcreate(0,0x10,10,&err);  /* create mesage queue */
    printf("P1: error code for create %d\n",err);
    printf("P1: pending for message .....\n");
    my_msg= sc_qpend(0x10,timeout,&err);   /* pending for message */
    printf("P1: message arrived error code %d\n",err);

    printf("P1: message is %s\n",my_msg);
    printf("P1: END OF task 1\n");
}
```

```
char * rqaccept();
char * sc_qpend();
main()
{
    int err;
    char *my_msg;


    printf("P4: task created\n");
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";
    my_msg=" ## Howdy ##";  /* delay */

    sc_rqcreate(0,0x22,10,&err);

    printf("P4: message at task 4 is >>>> %s\n",my_msg);

    printf("P4: Posting Message to task 1\n");
    sc_rqpost(1,0x10,my_msg,&err);     /* talk to task 1 */
    printf("P4: error code for post in task 4 %d\n",err);

    printf("P4: END of task 4\n");
}
```

```
################################ sandy -l tree1 tree2 tree4 ####################
P1: task created
P1: Sin value is 9.274369e-01
P1: error code for create 0
P1: pending for message .....
P2: task created
P2: error code for create 0
    >> P2: -- delaying 1.
    >> P2: -- delaying 2.
    >> P2: -- delaying 3.
    >> P2: -- delaying 4.
    >> P2: -- delaying 5.
    >> P2: -- delaying 6.
    >> P2: -- delaying 7.
    >> P2: -- delaying 8.
    >> P2: -- delaying 9.
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P2: error in message accept ..
P2: error code for accept 11
P2: END of task 2
P4: task created
P4: message at task 4 is >>>>  ## Howdy ##
P4: Posting Message to task 1
P1: message arrived error code 0
P1: message is  ## Howdy ##
P1: END OF task 1
P4: error code for post in task 4 0
P4: END of task 4

Speed up      0.974.
Efficiency    0.974.
```

```
#######################   sandy -2 tree1 tree2 tree4   #######################
P1: task created
P2: task created
P1: Sin value is 9.274369e-01
P2: error code for create 0
    >> P2: -- delaying 1.
    >> P2: -- delaying 2.
    >> P2: -- delaying 3.
    >> P2: -- delaying 4.
    >> P2: -- delaying 5.
P1: error code for create 0
    >> P2: -- delaying 6.
    >> P2: -- delaying 7.
P1: pending for message .....
    >> P2: -- delaying 8.
    >> P2: -- delaying 9.
P2: error 11
P4: task created
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P2: error in message accept ..
P2: error code for accept 11
P2: END of task 2
P4: message at task 4 is >>>>  ## Howdy ##
P4: Posting Message to task 1
P1: message arrived error code 0
P1: message is  ## Howdy ##
P1: END OF task 1
P4: error code for post in task 4 0
P4: END of task 4

Speed up      1.751.
Efficiency    0.842.
```

```
############################## sandy -3 tree1 tree2 tree4 ####################
P1: task created
P2: task created
P4: task created
P1: Sin value is 9.274369e-01
P2: error code for create 0
    >> P2: -- delaying 1.
    >> P2: -- delaying 2.
    >> P2: -- delaying 3.
    >> P2: -- delaying 4.
    >> P2: -- delaying 5.
P1: error code for create 0
    >> P2: -- delaying 6.
    >> P2: -- delaying 7.
P1: pending for message .....
    >> P2: -- delaying 8.
    >> P2: -- delaying 9.
P4: message at task 4 is >>>>  ## Howdy ##
P4: Posting Message to task 1
P2: error 11
P2: error 11
P2: error 11
P2: error 11
P1: message arrived error code 0
P1: message is  ## Howdy ##
P2: error 11
P1: END OF task 1
P4: error code for post in task 4 0
P2: error 11
P4: END of task 4
P2: error in message accept ..
P2: error code for accept 11
P2: END of task 2

Speed up      2.312.
Efficiency    0.771.
```

Part II

Allocation for the SANDAC Multiprocessor System

ALLOCATION FOR THE SANDAC MULTIPROCESSOR SYSTEM

T. M. Ravi and M. D. Ercegovac

UCLA Computer Science Department

Introduction

In this report we discuss the allocation of tasks in the SANDAC IV system ([BORG 83]). Initially we outline the model of execution and the underlying assumptions. We then discuss a graph reduction algorithm for preprocessing the computation graph, which is particularly necessary if the graph is very fine grain. The allocation algorithm is presented along with performance curves for different graphs. In the appendix, details of the software implementation and its use is discussed.

Model of Computation

The program is represented by a data flow graph ([DENN 80]), with nodes representing tasks and arcs representing precedence relationships between tasks. The partial ordering of the tasks necessary for correct execution is captured by the dependencies between these tasks. The nodes have a single point of entry and a single point of exit, i.e., a task can begin execution only when all its inputs (arguments) have arrived, and can deliver each of its results to destination tasks only after the execution of the task is completed. Likewise, the graph has a single entry node and a single exit

* This work has been supported in part by the Contract No. 25-3074 from the Sandia National Laboratories "Multiprocessor System Evaluation and Programming Environment"

node.

To represent control structures such as conditionals and loops in data flow graphs we introduce two special nodes (Figure 1). The "OR" node has three input arcs and one result arc. One of the arguments is boolean, and depending on its value, a token from one of its arcs (true or false arc) is processed and placed on the result arc. This special node is unlike other nodes which require all inputs to be present before the node can be activated.

The "SW" node has two input arcs, one being boolean; and two result arcs (True and False). Depending on the boolean value the result token is put on one of the result arcs. The "SW" and "OR" are in the same flavor as the Switch and Merge actors discussed by [DENN 80]. The "SW" operator on firing will output a token on either of its output arcs and the "OR" will fire when a token is present on any one of its input arcs.

Our present implementation of the allocation algorithm is for directed graphs without loops. Loops implemented by "SW" and "OR" operators could be handled by applying our algorithm in an hierarchical manner.

It is assumed that the execution time (t_p) of each node (tasks) is known apriori. There is a communication time (t_c) associated with each arc in the graph, whose value depends on the size of data communicated. Furthermore, the communication time can take on a lower value - local communication time (t_{cl}), or a higher value - bus communication time (t_{cb}). Bus communication time is chosen if results from one task have to be sent to another task in a different processor. Local

communication time is chosen when tasks reside in the same processor. One point to note is that the processors are busy during communication and will not become available until all the results are sent to their destinations. Results are sent out sequentially, due to limitations imposed by the communication mechanism, and hence the total communication time (t_c) is the sum of individual communication times of each result.

A task once started is not interrupted and will run till completion. A task can be activated only when all its arguments have arrived.

The objective is to allocate the tasks to a multiprocessor (given n processors), in order to obtain minimum execution times.

Graph Reduction

To reduce the complexity of the allocation process and to utilize the parallelism efficiently, we can reduce ([GAUD 84] & [ERCE 84]) the original graph into a larger grain task graph. By applying a set of rules, subgraphs in the data flow graph are replaced by a single node. The criterion for lumping together instructions into a single task is to minimize the response time for the subgraph under consideration.

When the delay incurred due to interprocessor communication and activation exceeds the gain in time due to concurrent execution, it is no longer justifiable to distribute the nodes over several processors. When the response time of a subgraph

executed sequentially in a single processor is less than or equal to the response time when executed concurrently, then the subgraph is reduced to a single node and is executed sequentially.

The condition ([RAVI 86]) for combining a node with its arguments is:

$$\sum_{i=1}^{narg} t_{parg} \leq \max_{i}(t_{parg} + t_{carg})$$

where t_{parg} is the processing time of the argument node

t_{carg} is the communication time of the argument node and

narg is the number of arguments.

If this condition is satisfied then the node and its argument nodes are lumped together into a single node.

This step is illustrated in Figure 2. Figure 2a is a subgraph where the nodes are separated in order to take advantage of the parallelism, while in Figure 2b the nodes A, B and C have been lumped together into a single node. In the subgraph of Figure 2a, node D can execute only after the results from node A and B and C have arrived. If nodes A, B and C are activated at the same time, then the result from nodes A and B will arrive after 5 cycles and the result from node C will arrive after 8 cycles. Hence node D is activated only after 8 cycles. In the sequential case the result from nodes A, B and C are available after 6 cycles, as we do not have to communicate between different processors. In this case the subgraph of Figure 2a can be reduced to Figure 2b.

```
Procedure main (G:typegraph);
{This procedure increases the grain size of the data flow graph (G).
 Starting at the root , nodes are combined with its arguments.}

begin
 UPREDUCTION(Root(G));
end;{main}


Procedure UPREDUCTION (i:typenode);
{This recursive procedure lumps a node and its arguments
 together, based on criterion depending on the the processing
 time and communication time. Each node has the the fields
 argument (arg), no. of arguments (narg), code (funct),
 processing time (proctime) and communication time
 (commtime).}

begin
 with node[i] do
  if narg > 0 then begin
  {test condition}
   seqtime:=0; partime:=0;
   for k:=1 to narg begin
     seqtime:=seqtime + node[arg[i]].proctime;
     if (node[arg[k]].proctime + node[arg[k]].commtime)
              > partime then
       partime := node[arg[k]].proctime + node[arg[i]].commtime;
   end;
   if ((partime -seqtime ≤ 0) or (if any arg has > than one result)) then
    {condition for reduction of parallelism is not true}
    for k:=1 to narg do UPREDUCTION(k);
   else begin
    {condition is false}
    copy the code in each of the arguments to node[i].funct
    node[i].proctime := seqtime;
    node[i].narg := sum of the narg of each of the arguments
                    of node[i]
     arguments of new node := arguments of all nodes combined
                with node[i]
    remove the old argument nodes from graph
    UPREDUCTION(i);
   end;
end;{UPREDUCTION}
```

Figure 3 : Upreduction Algorithm

Figure 1 : OR and SW operators

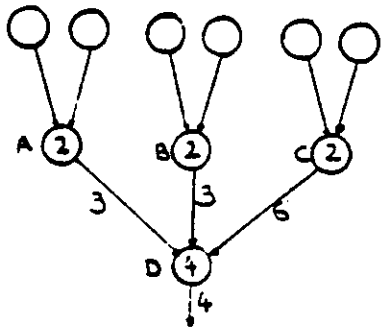

Figure 2a : Fine Grain Graph

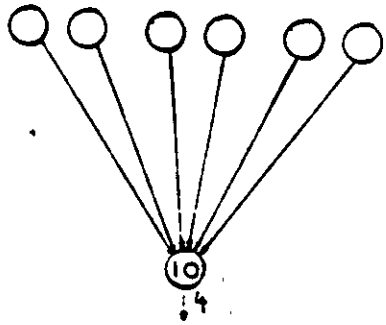

Figure 2b : Lumped Graph

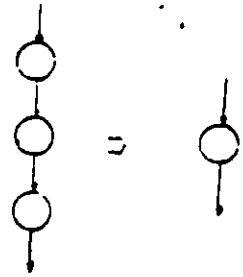

Figure 4 : Reduction of Sequential Nodes

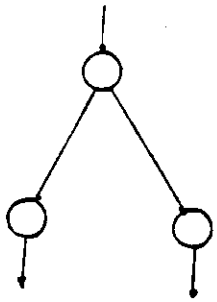

Figure 5 : Node with many results

The "upreduction" algorithm (Figure 3) spans the graph, testing criterion for reduction, in O(n) time. It combines a node with its arguments whenever the reduction criterion is met.

Note that sequential nodes which have single arguments and single results are combined together into a single node (Figure 4). Execution of each of the sequential nodes in a different processor leads to unnecessary overhead.

However, when a node has more than one result which goes to different nodes, then it can not be combined by the "upreduction" algorithm. In order to reduce these subgraphs (Figure 5), a "downreduction" algorithm has to be applied with the entry node as a parameter. It combines a node and its results based on the processing time and communication time criterion into a single node. The algorithm is similar to the "upreduction" algorithm.

The graphs of Figure 6a,6b & 6c illustrate the Graph Reduction algorithm, with an example of an iteration consisting of 30 nodes (Figure 6a), which also has a conditional statement in it. After a single pass of the reduction algorithm, i.e., combination of a node and its arguments, we obtain a graph with 19 nodes (Figure 6b). After another pass of the reduction algorithm , i.e., combination of a node and its results, we obtain the final reduced graph consisting of 13 nodes (Figure 6c). We are now ready to allocate this graph to the processors.

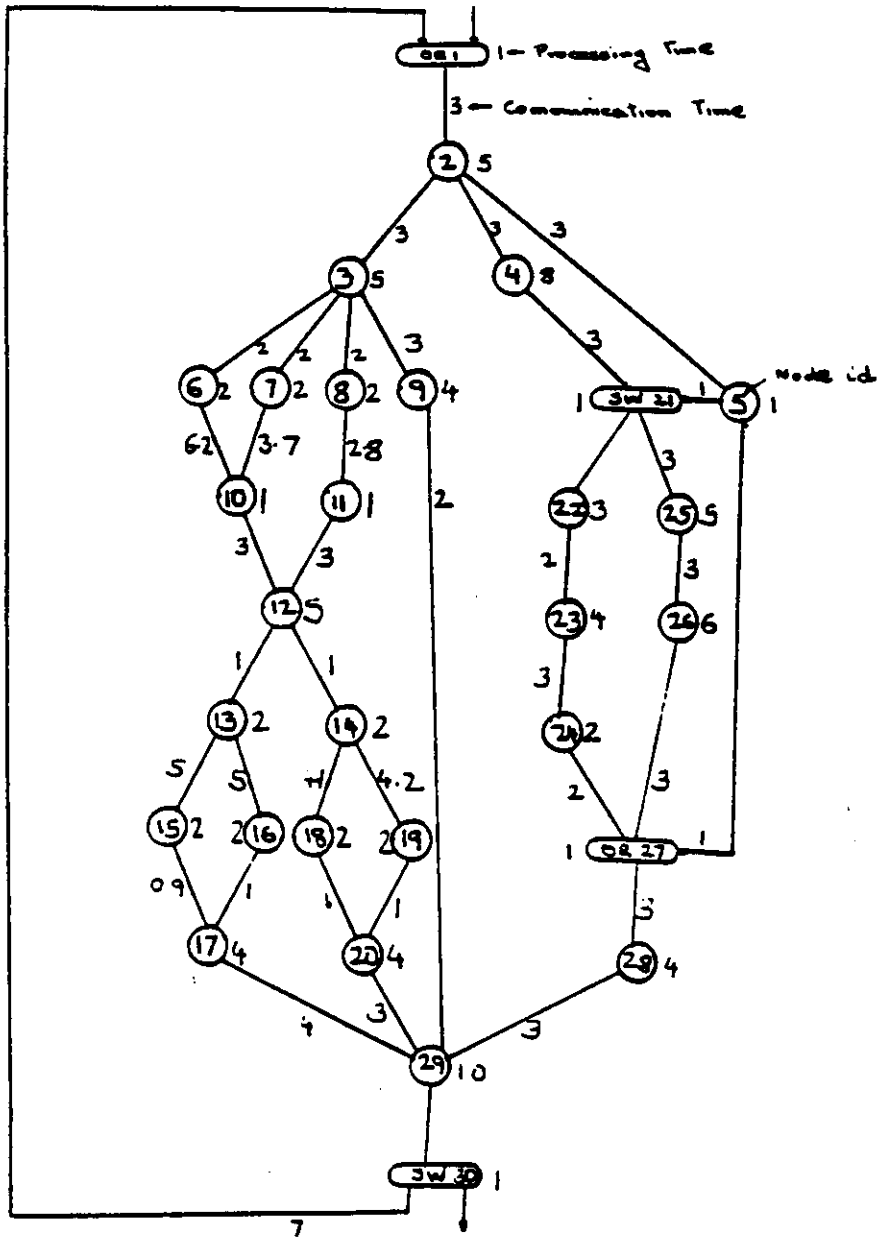

Figure 6a : Initial Data Flow Graph (30 nodes)

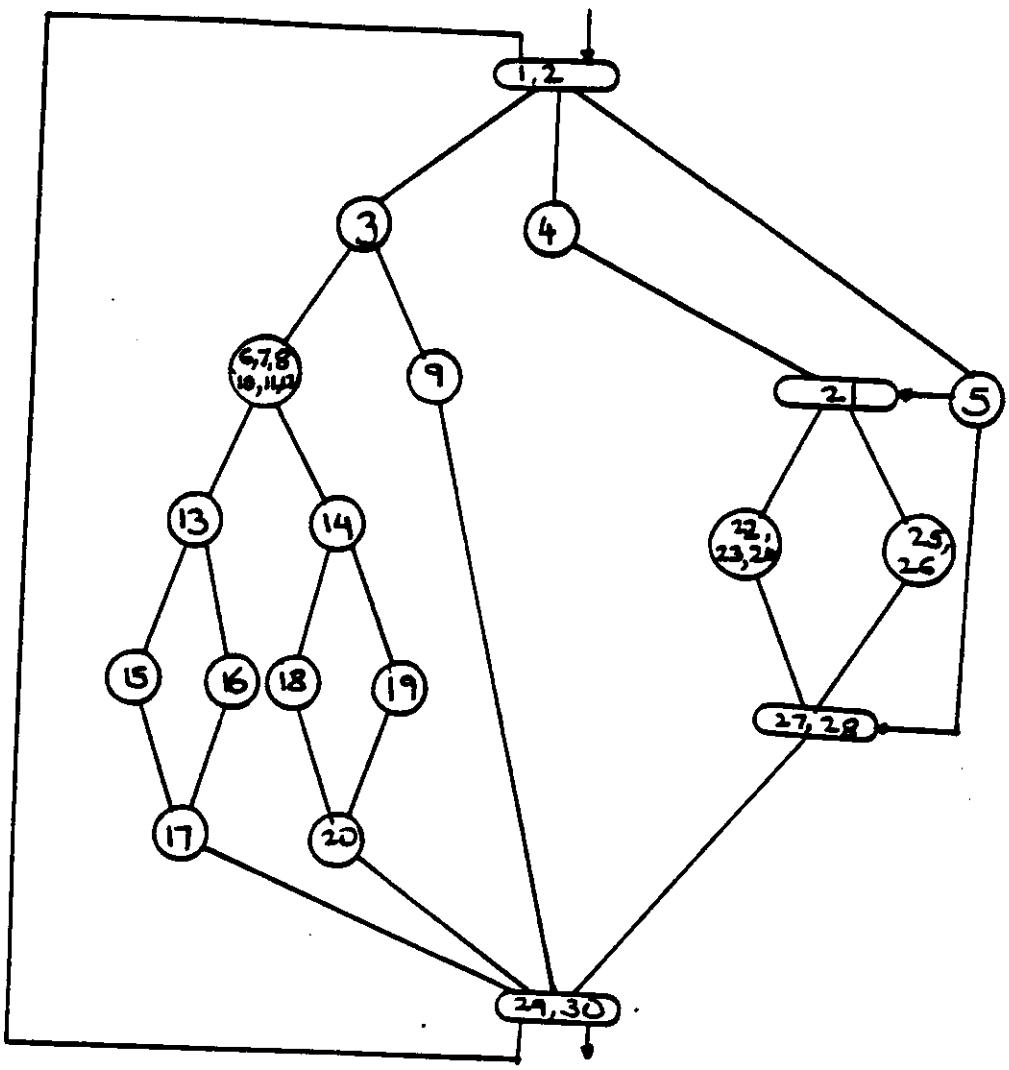

Figure 6b : Intermediate Graph After Upward Reduction (19 nodes)

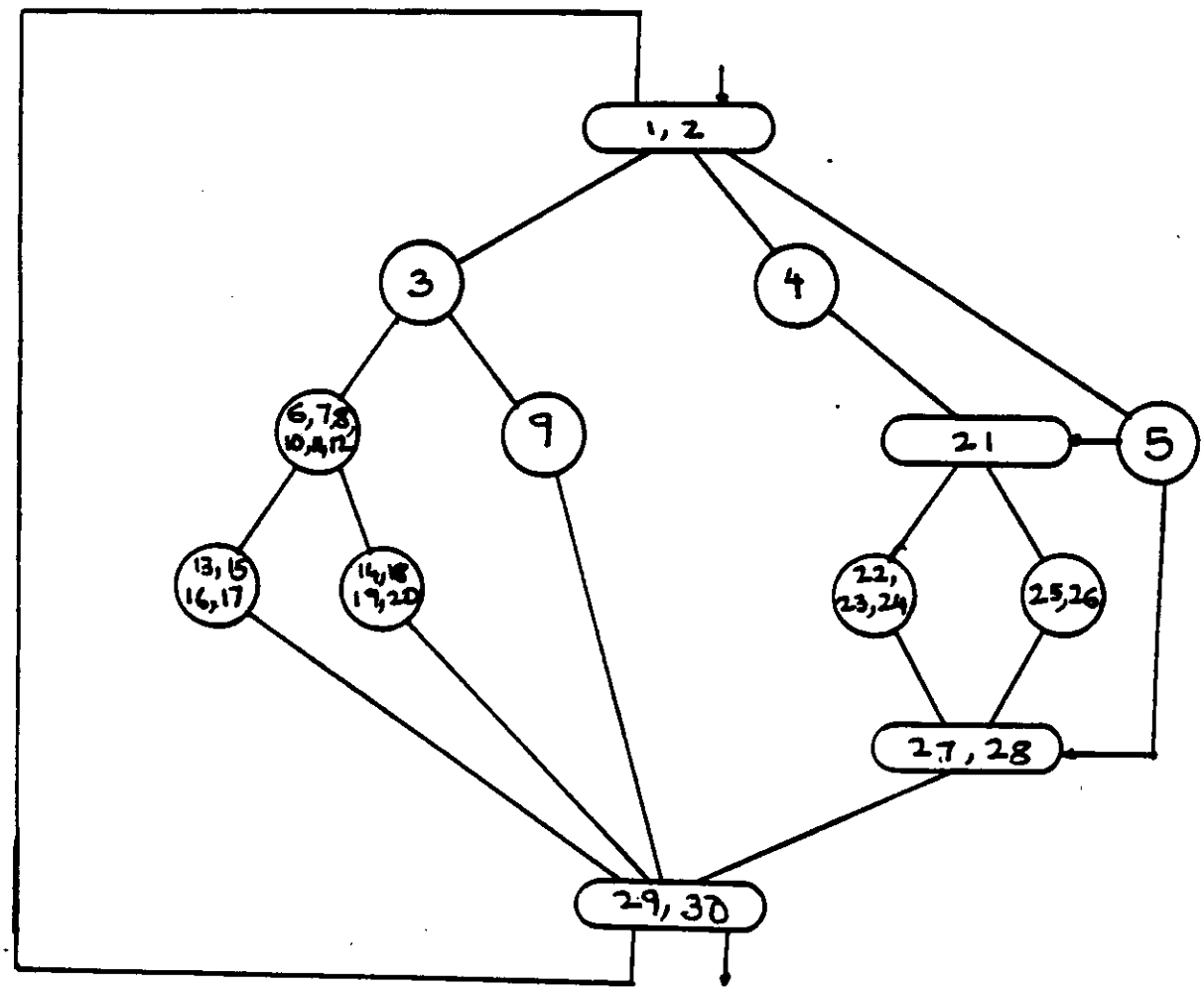

Figure 6c : Final Graph After Downward Reduction (13 nodes)

Task Allocation

The heuristic allocation algorithm minimizes response time based on two principles :

1) Precedence to critical tasks

2) Minimizing communication time between tasks

An allocation algorithm based on the first principle of critical path scheduling, when the tasks only have processing times associated with them is discussed in [KOHL 75]. The second principle of minimizing communication time provides a criterion for selecting a task for allocation when several candidates are available. It enables us to allocate predecessor-successor tasks to the same processor, thus incurring the lower local communication time.

The difficulty in applying the critical path algorithm to this problem is that timing parameters associated with the graph cannot be fixed until the allocation is itself complete. This is because the decision on whether to choose local or bus communication time for an arc depends on where the successor task will be allocated. This leads to two specific problems. First, critical paths which are the longest paths in the graph cannot be precisely determined. Second, when a task is allocated to a processor, we cannot determine exactly when the task will complete, because it is not known at that stage in the algorithm as to where the successor tasks are going to be allocated in order to choose the right communication time. In our algorithm we show how these two problems can be handled.

The Algorithm

Consider a graph with tasks $T_1, T_2 \ldots T_k$, to be executed on n processors P_1, $P_2 \ldots P_n$. Two lists are constructed - Processor list (L_p) and Task list (L_t). The processor list, at any stage of the algorithm, contains the processors listed in increasing order of busy times, i.e., the time up to which they are busy. The processor on the top of the list is the one which will become free next. Initially, the processors are in random order in the list, as they are free. The task list is generated based on critical path lengths. The critical path length ($CP(T_i)$) of a task T_i, is defined to be the length of the longest path from the exit node to T_i. To calculate the critical paths, we assume that the value of the communication time taken for each arc is the higher bus time. The critical paths of nodes in a graph are calculated starting from the exit node. The critical path of the exit node is equal to t_p+t_c, where t_c is the sum of the bus communication times of all the results. The critical path of any other node in the graph is equal to the maximum critical path of result nodes + t_p + t_c, where again t_c is the sum of the bus communication times of all the results. The task list (L_t) is generated by sorting the tasks in decreasing order of their critical paths. At any stage of the algorithm, the list contains tasks yet to be allocated.

At any time we choose the top processor from the processor list (L_p), which is the first to become idle. The task list is then scanned till we can choose the first candidate for execution in the processor. Any other task on the list which can be executed, and is within a deviation of Δ from the critical path of the first candidate, is also chosen as a candidate. A task can be a candidate only if at the time when the processor becomes free all its arguments have arrived, i.e., all its predecessors have

completed execution.

Now we choose the task among the candidates to be assigned to the processor. Of all the candidate tasks, we choose the task which when allocated to the processor gives the maximum saving in communication time. A saving in communication time is made if the predecessor tasks are assigned to the same processor. The saving is the sum of the difference of the bus communication time and local communication time for each direct predecessor assigned to the same processor.

The chosen task is assigned to a processor, but the question that arises is - What will the duration of the execution of this task be ? This would be t_p+t_c, but we don't know whether to take the local or bus communication time for the results of the task, as the successor tasks have not yet been allocated.

The solution to this problem is to associate communication times with arguments instead of results. Thus, when a task is allocated, the location of its predecessor is known. In our model the communication of the results is the responsibility of the task, and to take care of this we reverse the graph. The direction of the arcs in the graph is reversed before the calculation of critical paths and the generation of lists. On starting with the reversed graph, the schedule obtained can be reversed to obtain a regular allocation. By reversing the graph, the communication time of the arcs is associated with arguments to tasks and not results.

After the task has been assigned to the processor, the busy time of the processor is updated. The task is removed from the task list (L_t) and the processor is reinserted in the appropriate position in the processor list (L_p), which is ordered

according to increasing busy time.

If no task can be assigned to the processor (P_1), then we have to move to the time of the next event and try again. The processor list is scanned; and the first processor (P_2) with busy time greater than the busy time of this processor (P_1) is placed on the top of the list. Processor P_1 and any other processors with busy time equal to that of P_1 are updated with busy time equal to the busy time of P_2. In this way idle times are caused in processors when no tasks are ready.

This process of allocating each task to a processor continues till the task list is exhausted. The allocation algorithm is given in Figure 7.

```
procedure SELCANDIDATES(var candidate, var nocandidates,Δ,listsize);

{This procedure selects tasks which can be executed next on the processor L_p[1]}

begin
  i:=1; nocandidates:=0;
  while ((i =< listsize) and (nocandidates=0)) do begin
    if ((T[k].completion-time =< P[L_p[1]].busytime for all argument
       tasks (k) of task L_t[i])
     or (T[L_t[i]].narg = 0)) then begin
       candidate[1] :=L_t[i];
       nocandidates:=1;
    end else i:= i+1;
  end;

  if nocandidates > 0 then begin
    i:=i+1;
    limit:=T[candidate[1]].CP - Δ;
    while i =< listsize do begin
      if ((T[k].completion-time =< P[L_p[1]].busytime for all
         argument tasks (k) of task L_t[i])
       or (T[L_t[i]].narg = 0)) then begin
         candidate[1] :=L_t[i];
         nocandidates:=1;
       end;
       i:= i+1;
    end;
  end;
end; {SELCANDIDATES}
```

Figure 7c : Allocation Algorithm (Selection of Candidate Tasks)

Performance

To study the performance of the algorithm, several program graphs were allocated and statistics collected. The effect of changing the parameter Δ, which is the deviation in critical path for the choice of candidates, and the behavior of the algorithm for different ratios of processing time and communication time, were studied.

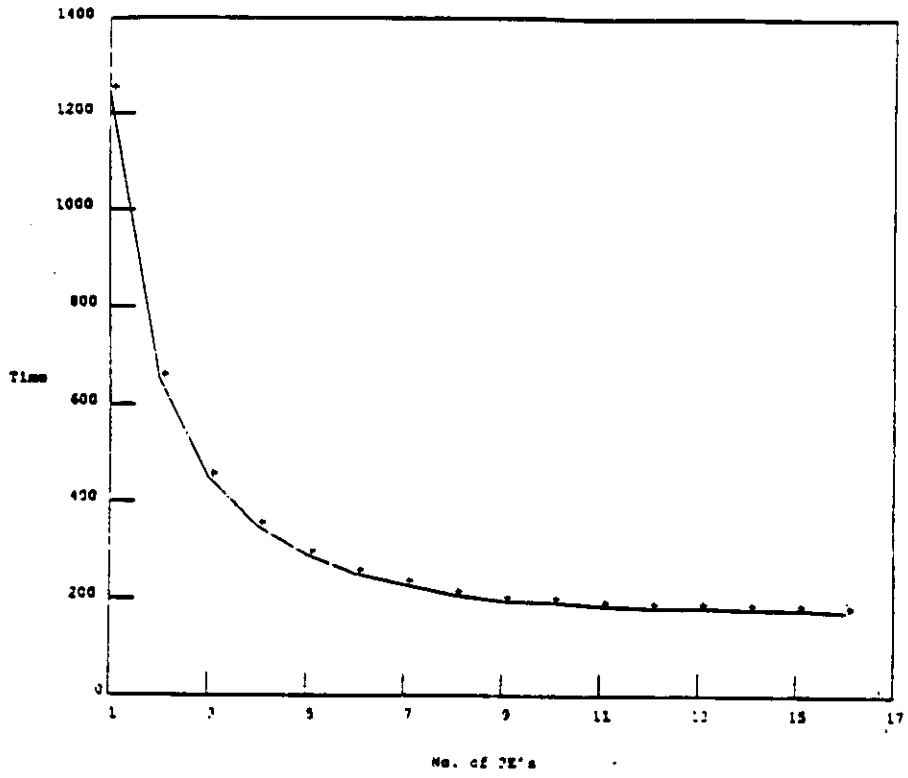

Figure 8 : Response Time (T) Vs. No. of Processors (N)

We first examine the speedup achieved by using multiprocessors. Figure 8 shows the variation of response time (T) with the number of processors (N) for a graph (Figure 9) containing 123 nodes. The processing time of each node is 20, the local communication time 0.1, the bus communication time 1 and the deviation (Δ) 1 unit of time. We observe that initially when the amount of concurrency exceeds the number of processors available, the response time falls rapidly with the increase in the number of processors. Figure 10 illustrates the speedup (T[1]/T[i]) of the multiprocessor system over a single processor. With a multiprocessor system

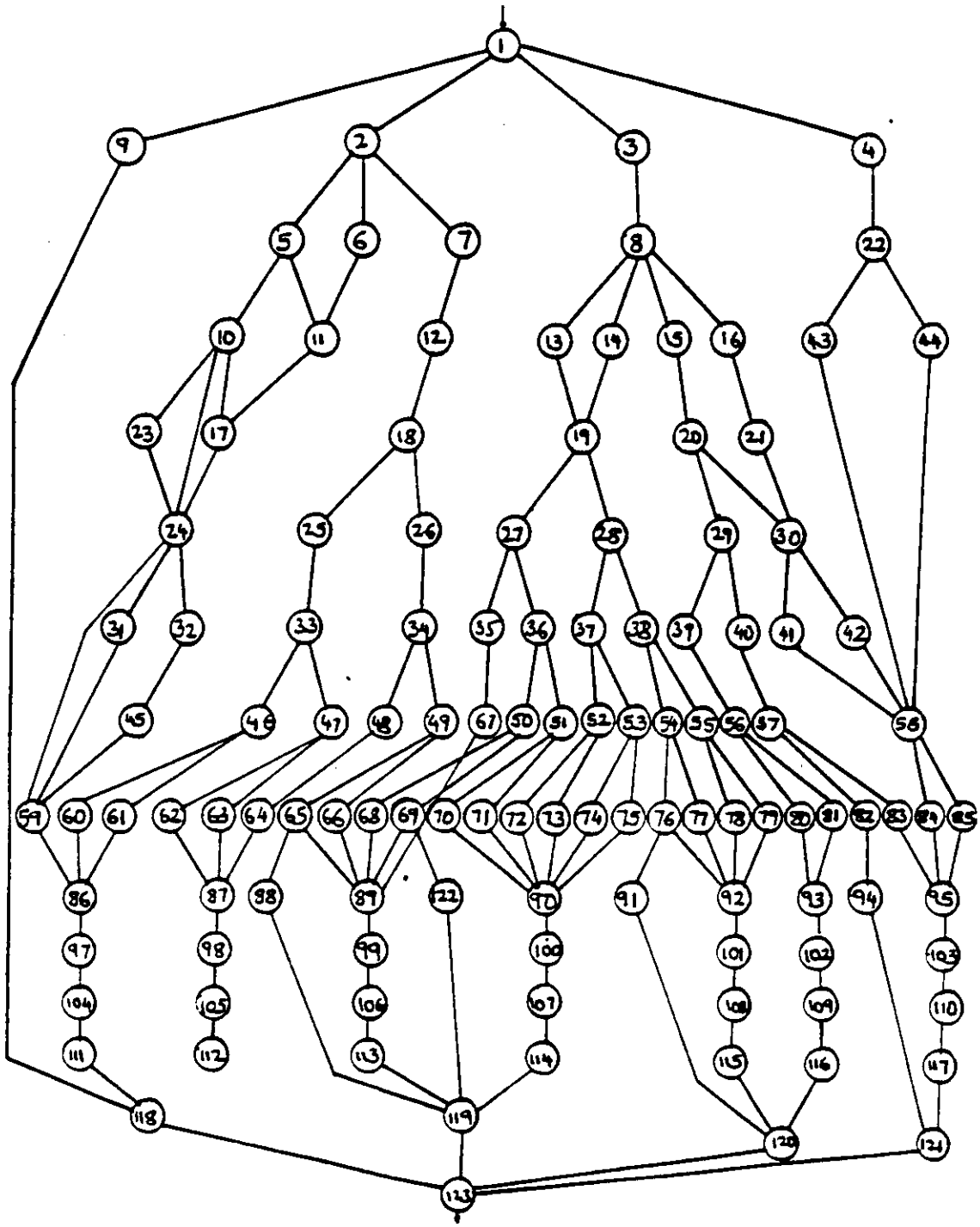

Figure 9 : Graph With 123 Nodes

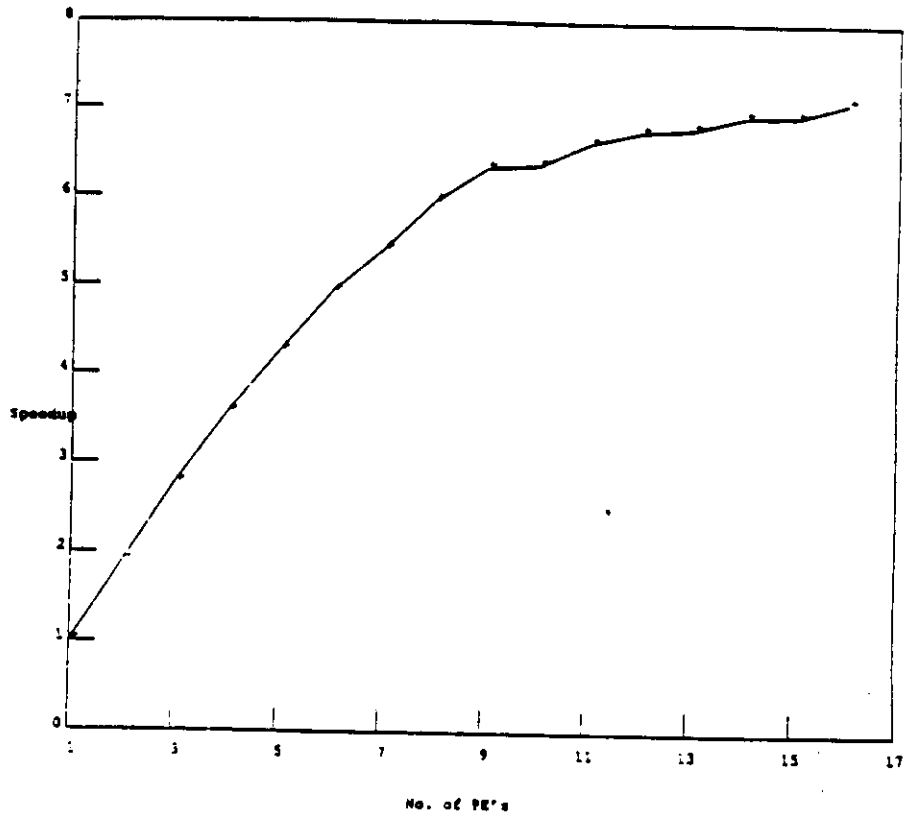

Figure 10 : Speedup Vs. No. of Processors

consisting of 8 processors, the speedup over the uniprocessor is 6. Initially, when the number of processors is increased the speedup is almost linear, but as the amount of concurrency is exhausted the curve saturates. Figure 11 demonstrates the efficiency (Speedup/N) of the processors in the multiprocessor system. The fall in efficiency is attributed to the dependencies in the graph which force idle times in some processors when very few tasks can be activated.

The algorithm has two driving principles - Precedence to critical tasks (critical path scheduling) and the minimization of communication time between tasks. Figure

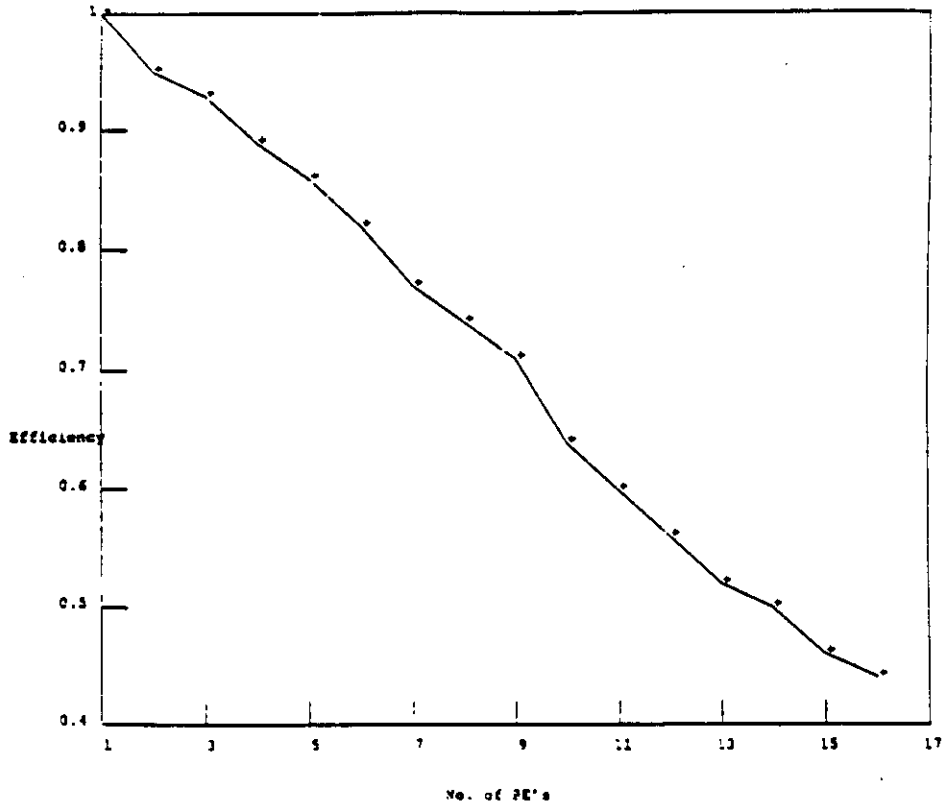

Figure 11 : Efficiency Vs. No. of Processors

12 shows the performance when only critical path scheduling is enforced. The example is of a sort-merge graph (Figure 13) with 94 nodes, where the processing time of each node is 20 units, the local communication time is 0.1 units and the bus communication time is 5 units. The curve (a) shows the response time for a strict list schedule where no attempt is made to have predecessor-successor tasks cohabit in the same processor. Curve (b) uses our algorithm with a deviation (Δ) equal to 0.1, which is the local communication time. The deviation (Δ) is usually chosen to be a factor of the bus communication time. For two processors the difference in the response times is 15%, due to the large saving from the reduced interprocessor communication.

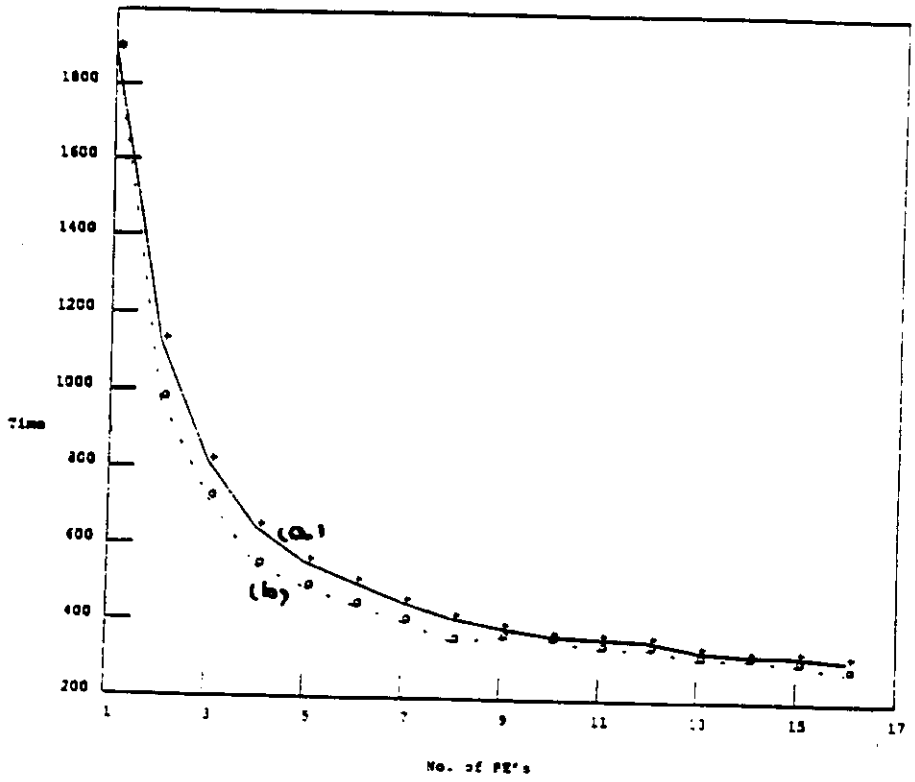

Figure 12 : Comparison of Performance with Critical Path List Schedule

When the deviation is very large, i.e., several orders of magnitude larger than the bus communication times, then the critical path list ordering is no longer operative. In Figure 14 we have a program graph with one dominant critical path and several non-critical tasks. When the deviation exceeds the length of the critical path, then at each stage the candidates for allocation to a processor are all the enabled tasks in the graph. In other words critical and non critical tasks are given equal chance for execution at any point. For two processors for the graph of Figure 14, with t_p =20, t_{cb} = 1 & t_{cl} = 0.1 we observe that the response time increases by 22% from zero

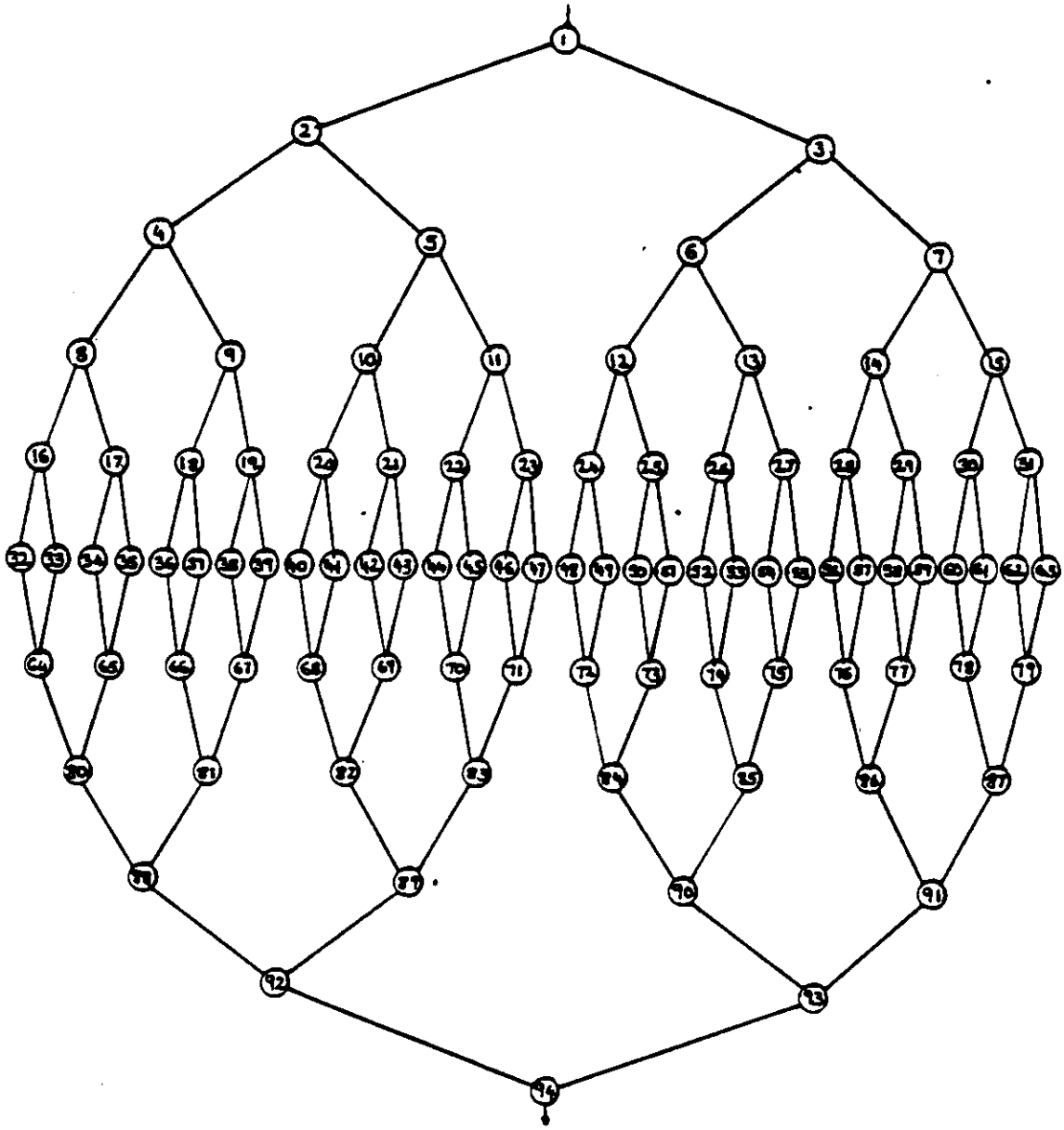

Figure 13 : Sort-Merge Graph

deviation response time, when the deviation is greater than the critical path.

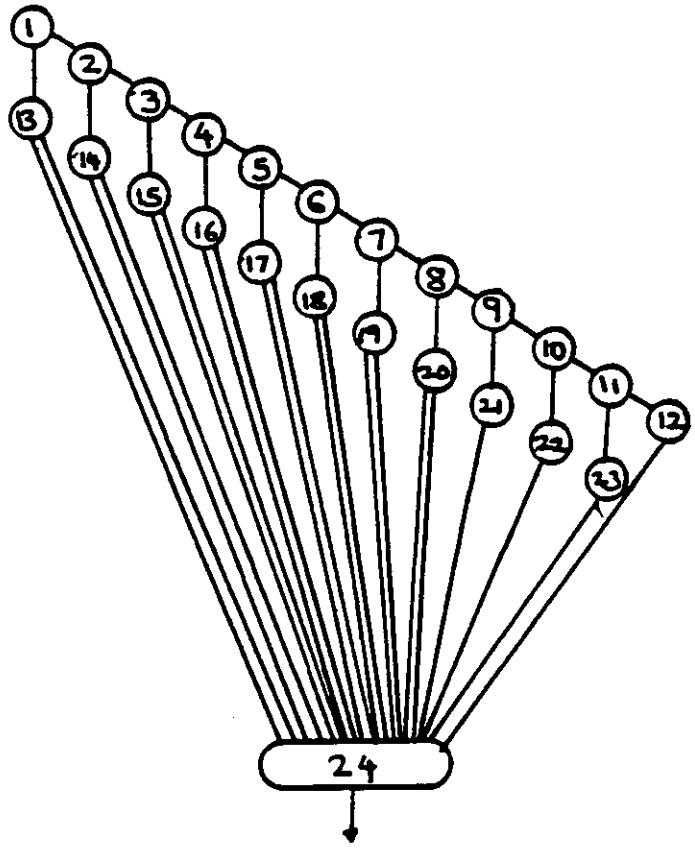

Figure 14 : Example with a Dominant Critical Path Schedule

A Variation to the Allocation Algorithm

One variation to the Allocation algorithm which we have considered is to evaluate critical paths based on the processing time alone. The motivation behind this variation (A_{CP}) to the algorithm is that here the communication times (bus or local communication times) will not influence the order of tasks in the critical path list. Our observation with the example (Figure 9) with 123 nodes shows that when the bus

Reference

[BORG 83] Borgman, C. R. and P. E. Pierce, "A Hardware/Software System for Advanced Development Guidance and Control Experiments," *AIAA Computers in Aerospace Conference*, AIAA-83-2416, Oct. 1983, Hartford, CT, pp. 377-384.

[DENN 80] Dennis, J. B., "Data Flow Supercomputer Languages," *Computer*, Nov. 1980, pp. 48-56.

[ERCE 84] Ercegovac, M. D., P. K. Chan and T. M. Ravi, "A Dataflow Multiprocessor Architecture for High Speed Simulation of Continous Systems," *Proc. International Workshop on High-Level Architecture, 1984.*

[GAUD 84] Gaudiot, J. L., and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," *Proc. 4th International Conf. on Distributed Computing Systems*, 1984, pp. 2.9-2.17.

[KOHL 75] Kohler, Walter H., "A Preliminary Evaluation of Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, Vol. C-24, Dec. 1975, pp. 1235-1238.

[RAVI 86] Ravi, T. M., "Partitioning and Allocation of Functional Programs for Data Flow Processors," *M.S. Thesis*, UCLA Computer Science Department, Feb. 1986.

Appendix 1

The software tools for the allocation of tasks to the SANDAC IV architecture consists of two programs - allocation.p and reduction.p, implemented in Berkeley Pascal and given in Appendices 2 and 3 respectively. The input file to reduction.p is 'ingraph'. Appendix 1.1 shows the format of ingraph for the graph of Figure 5a. The main program allocation.p has input file 'outgraph2' if the original graph is to be allocated and 'outgraph4' (Appendix 1.2) if the reduced graph is to be allocated. Files 'outgraph2' and 'outgraph4' are output files from reduction.p. A session illustrating the execution of the programs is given below.

```
<1> reduction.out
Want to parameterize Communication and Processing times (y or n) ?y
Processing time:10
Local Communication time:0.5
Bus Communication time:5
Pr.time =  1.00e+01
LocComm.time =  5.00e-01
BusComm.time =  5.00e+00
<2> allocation.out
Input is original graph {o} or reduced graph {r} r
Reading reduced graph
The critical path of the graph is   185.000
No. of processors =2
 Deviation in critical path for selecting candidates =0
 Deviation in critical path is =     0.00
The response time for       2 processors is   206.500
```

The output files of allocation.p are 'outstat' and 'outschdr'. File 'outschdr' (Appendix 1.3) lists the tasks assigned to each processor and 'outstat' gives statistics on the allocation.

APPENDIX 1.1

```
{Input Computation Graph - ingraph}
This is the input file to program
reduction.p. This graph is reduced
based on the reduction criterion to
obtain a large grain graph.

#1
1       {Node id. or code}
1.0     {Processing time}
        {Argument Nodes}
2       {Result Nodes}
3.0     {Local Communication Time}
6.0     {Bus Communication Time}

#2
2
5.0
1
3 4 5
2.0 1.5 2.0
6.0 4.0 3.0

#3
3
5.0
2
6 7 8 9
2.0 2.0 2.0 3.0
4.0 4.0 3.0 3.7

#4
4
8.0
2
21
3.0
3.8

#5
5
1.0
2
21 27
1.0 1.0
3.0 2.0

#6
6
2.0
3
10
1.2
2.2

#7
7
2.0
3
10
1.7
3.7

#8
8
2.0
3
11
0.8
2.8

#9
9
4.0
3
29
1.0
2.0

#10
10
1.0
6 7
12
1.0
2.0

#11
11
1.0
8
12
1.0
2.0

#12
12
5.0
10 11
13 14
1.0 1.0
2.0 1.8

#13
13
2.0
12
15 16
2.0 1.0
5.0 2.0

#14
14
2.0
12
18 19
0.1 2.2
1.1 4.2

#15
15
2.0
13
17
0.9
1.9
```

```
#16
16
2.0
13
17
1.0
2.0

#17
17
4.0
15 16
29
3.0
6.0

#18
18
2.0
14
20
1.0
4.0

#19
19
2.0
14
20
1.0
2.0

#20
20
4.0
18 19
29
1.0
2.4

#21
21
1.0
4 5
22 25
1.0 3.0
2.0 4.0

#22
22
3.0
21
23
2.0
3.0

#23
23
4.0
22
24
1.0
3.0

#24
24
2.0
23
27
2.0
3.0

#25
25
5.0
21
26
1.0
3.0

#26
26
6.0
25
27
3.0
4.0

#27
27
1.0
5 24 26
28
1.0
3.0

#28
28
4.0
27
29
2.0
3.0

#29
29
10.0
17 20 9 28
30
2.0
5.0

#30
30
1.0
29
```

APPENDIX 1.2

```
{Reduced graph - outgraph4}
This is the output of reduction.p
after the graph is parameterized
and then reduced. This graph is
the input to the allocation
program allocation.p.

NO OF NODES        23
node          1
funct              2,1
proctime                  20.000
narg                       0
nres                       3
{results}     2       3       4
{loc comm}  0.500   0.500   0.500
{bus comm}  5.000   5.000   5.000

node          2
funct              3
proctime                  10.000
narg                       1
{arg node}    1
nres                       4
      5       6       9       7
  0.500   0.500   0.500   0.500
  5.000   5.000   5.000   5.000

node          3
funct              4
proctime                  10.000
narg                       1
        1
nres                       1
       19
    0.500
    5.000

node          4
funct              5
proctime                  10.000
narg                       1
        1
nres                       2
       19      22
    0.500   0.500
    5.000   5.000

node          5
funct              6
proctime                  10.000
narg                       1
        2
nres                       1
        8
    0.500
    5.000

node          6
funct              7
proctime                  10.000
narg                       1
        2
nres                       1
        8
    0.500
    5.000

node          7
funct              9
proctime                  10.000
narg                       1
        2
nres                       1
       23
    0.500
    5.000

node          8
funct             10
proctime                  10.000
narg                       2
        5       6
nres                       1
       10
    0.500
    5.000

node          9
funct             11,8
proctime                  20.000
narg                       1
        2
nres                       1
       10
    0.500
    5.000

node         10
funct             12
proctime                  10.000
narg                       2
        8       9
nres                       2
       11      12
    0.500   0.500
    5.000   5.000

node         11
funct             13
proctime                  10.000
narg                       1
       10
nres                       2
       13      14
    0.500   0.500
    5.000   5.000

node         12
funct             14
proctime                  10.000
narg                       1
       10
nres                       2
       16      17
    0.500   0.500
    5.000   5.000
```

```
node          13
funct               15
proctime                    10.000
narg                         1
        11
nres                         1
        15
     0.500
     5.000

node          14
funct               16
proctime                    10.000
narg                         1
        11
nres                         1
        15
     0.500
     5.000

node          15
funct               17
proctime                    10.000
narg                         2
        13        14
nres                         1
        23
     0.500
     5.000

node          16
funct               18
proctime                    10.000
narg                         1
        12
nres                         1
        18
     0.500
     5.000

node          17
funct               19
proctime                    10.000
narg                         1
        12
nres                         1
        18
     0.500
     5.000

node          18
funct               20
proctime                    10.000
narg                         2
        16        17
nres                         1
        23
     0.500
     5.000

node          19
funct               21
proctime                    10.000
narg                         2
        3         4
nres                         2
        20        21
     0.500     0.500
     5.000     5.000

node          20
funct               24,23,22
proctime                    30.000
narg                         1
        19
nres                         1
        22
     0.500
     5.000

node          21
funct               26,25
proctime                    20.000
narg                         1
        19
nres                         1
        22
     0.500
     5.000

node          22
funct               28,27
proctime                    20.000
narg                         3
        4         20        21
nres                         1
        23
     0.500
     5.000

node          23
funct               30,29
proctime                    20.000
narg                         4
        15        18        7         22
nres                         0
```

APPENDIX 1.3

(Allocation of tasks to processors - outschdr)
This output file from allocation.p indicates
which tasks are allocated to which processors.
It also gives the starting time and finishing
time for the tasks when executed in the
reverse schedule.

		Task #	Start	Finish Time
PROCESSOR	1:	23(	0.0,	20.0)
		18(	20.0,	30.5)
		17(	30.5,	41.0)
		16(	41.0,	51.5)
		12(	51.5,	62.5)
		22(	62.5,	83.0)
		20(	83.0,	113.5)
		9(	113.5,	138.5)
		7(	138.5,	149.0)
		2(	149.0,	170.0)
No of tasks	10			
PROCESSOR	2:	15(	20.0,	35.0)
		14(	35.0,	45.5)
		13(	45.5,	56.0)
		11(	56.0,	67.0)
		10(	67.0,	82.5)
		8(	82.5,	93.0)
		21(	93.0,	118.0)
		6(	118.0,	128.5)
		5(	128.5,	139.0)
		19(	139.0,	154.5)
		4(	154.5,	170.0)
		3(	170.0,	180.5)
		1(	180.5,	206.5)
No of tasks	13			

APPENDIX 2

```
{*************************************************************************
        REDUCTION                   8/18/1985                T.M.RAVI
                            (c) by T. M. Ravi
                                  1985
 ************************************************************************* }
program reduction(input, output);
{
        This program reads in a program graph and reduces it based on
        communication and processing time criterion alone. We assume
        that the input graph is a single input-single output graph.

        INPUT:
              files   ingraph -  Program graph given by user
        OUTPUT:
              files   outgraph1 - Original graph without reduction
                      outgraph2 - Original graph with parameterized timing
                                  if parameterization option been excercised
                      outgraph3 - Graph after upward reduction
                      outgraph3 - Graph after downward reduction
                                  Final reduced graph
        PROCEDURE:
              upreduc          - Reduces the graph starting at the result node
              dnreduc          - Reduces the graph starting at the entry node
              datain           - Inputs the graph from file ingraph
              dataout          - Prints the current graph
              remnodes         - Removes nodes from tree  structure which are
                                 no longer present
              parameterize     - Allows parameterization of processing time,
                                 local and bus communication time.
}
const
    maxnodes = 130;              {maximum number of nodes in program graph}
    maxfunchar = 100;            {maximum characters in definition of function}


type
    tmaxnodes= 0..maxnodes;
    tfunct= packed array[1..maxfunchar] of char;
    targ=^link1;
    link1= record
            no:tmaxnodes;                {index of argument node      }
            dir:char;                    {arg label,f-forward arc,b-backward arc}
            next:targ;                   {pointer to next arg         }
        end;
    tres=^link2;
    link2= record
            no:tmaxnodes;                {identifier of the node              }
            dir:char;                    {res label,f-forward arc,b-backward arc}
            commtime:real;               {communication time of result arc    }
            bustime:real;                {bus communication time of result arc }
            next:tres;                   {pointer to next res         }
        end;
    tnode = record                        {structure for representation of     }
                                          {each node belonging to the graph    }
              funct: tfunct;              {description of node                 }
              narg: integer;              {number of arguments                 }
              arg: targ;                  {pointer to arguments                }
              nres: integer;              {number of results                   }
              res: tres;                  {pointer to results                  }
              proctime: real;             {processing time                     }
            end;
    typetree = array [1..maxnodes] of tnode; {tree = collection of nodes     }
```

```
var

    tree: typetree;                    {array to store program graph   }
    nonodes: tmaxnodes;                {total number of nodes initially }
    newnonodes: tmaxnodes;             {total number of nodes  }
    entrynode: tmaxnodes;              {index of entry node}
    out:text;                          {var for text files }


{*****************************************************************************
                                                        DATAIN
 ***************************************************************************** }
procedure datain(var tree:typetree;var nonodes:tmaxnodes);
{

        Procedure to input the program graph from file ingraph.
        Ingraph has the nodes listed in order. An example of a node:

                #2                      (delimiter between nodes)
                2OR                     (node index 2 with function OR)
                1.0                     (Processing time)
                1 30b                  (arg. 1 and 30 with backward arc from 30)
                3 4                     (Result nodes)
                1.5 1.8                 (Local communication times for results)
                4.1 4.4                 (Bus comm. times)

                #3                      (next node ....)


        INPUT:  file ingraph

        OUTPUT:
                tree     -  tree (graph) as an array of nodes.
                nonodes  -  no. of nodes in initial graph.
}
var     i,j,l: integer;
        p:real;
        inp:text;
        tmpchar:char;
        firstptr,ptr,prevptr:targ;
        firstqtr,qtr,prevqtr:tres;
begin
  reset (inp,'ingraph');
  nonodes:=0;
  while not eof(inp) do begin
    read(inp,tmpchar);
    if tmpchar<> '#' then
      writeln('ERROR 1 in DATAIN - New node description should start with #')
    else begin
      nonodes:=nonodes+1;
      readln(inp,j);   {index of new node}
      with tree[j] do begin
                              {funct[1] & funct[2] are reserved. The function
                               starts from funct[3]}
        funct[1]:='U';i:=3;   {funct[1] can be 'X', 'D' or 'U'}
                              {'X' indicates that the node no longer exists &
                               'D' & 'U' are for book-keeping purposes}
        while not eoln(inp) do begin
          read(inp,tmpchar);
          funct[i]:=tmpchar;  {read the function and place it starting funct[3]}
          i:=i+1;
        end;
        funct[i]:=' ';
```

```
  {If 1st char. of function is 'S' then the function is SWITCH, if it is
   'O' then the function is 'OR' else the function is neither ('N').
   funct[2] is used to indicate whether a function is a SW,OR or neither}

 if funct[3]='O' then funct[2]:='O' else
   if funct[3]='S' then funct[2]:='S' else
   funct[2]:='N';
 readln(inp,proctime);                          {processing time of node}

 {read arguments of this node. The arguments are stored in a linked list}
 l:=0;i:=1; {l counts the no. of aruments}
 while ((not eoln(inp)) and (i<>0)) do begin
   read(inp,i);
   if i <> 0 then begin
     l:=l+1;
     new(ptr);
     if l=1 then firstptr:=ptr else prevptr^.next:=ptr;
     ptr^.no:=i;
     ptr^.next:=nil;
     prevptr:=ptr;
     if not eoln(inp) then begin
       read(inp,tmpchar);
       {if the arg. is a backward arc, i.e., coming from below this node
        (possible only for an OR node) then the input should indicate it
        example  30b indicates that the argument node is no. 30 and the
        arc from 30 to this node is a backward arc}

       if tmpchar='b' then ptr^.dir:='b' else
       if tmpchar<>' ' then writeln('ERROR 2 in DATAIN')
          else ptr^.dir:='f';  {direction is forward if not backward}
     end;
   end;
 end;
 readln(inp);
 if l<>0 then arg:=firstptr;
 narg:=l;

 {read in the result nodes}
 l:=0;i:=1;
 while ((not eoln(inp)) and (i<>0)) do begin
   read(inp,i);
   if i <> 0 then begin
     l:=l+1;
     new(qtr);
     if l=1 then firstqtr:=qtr else prevqtr^.next:=qtr;
     qtr^.no:=i;
     qtr^.next:=nil;
     prevqtr:=qtr;
     if not eoln(inp) then begin
       read(inp,tmpchar);
       if tmpchar='b' then qtr^.dir:='b' else
       if tmpchar<>' ' then writeln('ERROR 3 in DATAIN')
          else qtr^.dir:='f';
     end;
   end;
 end;
 readln(inp);
 if l<>0 then res:=firstqtr;
 nres:=l; {set nres=the counter l}

 {local and bus communication time are read from input graph. They will
  not be used if the parameterize option is chosen by the user }

 {read in the local communication time for each result. Note that for
```

```
          each result the input should have a corresponding local
          communication time. }
        qtr:=res;
        if nres > 0 then begin
          for l:=1 to nres do begin
            read(inp,p);
            qtr^.commtime:=p;
            qtr:=qtr^.next;
          end;
          readln(inp);
        end;
        {read in the bus communication time for each result}
        qtr:=res;
        if nres > 0 then begin
          for l:=1 to nres do begin
            read(inp,p);
            qtr^.bustime:=p;
            qtr:=qtr^.next;
          end;
          readln(inp);
        end;
        readln(inp);
      end;
    end;
  end;
end;    {datain}

{*****************************************************************************
                                                        REMNODES
 ***************************************************************************** }
procedure remnodes( var tree:typetree;var nonodes:tmaxnodes);
{

         Procedure to remove nodes which no longer exist (i.e., that have been
         combined). Basically to clean up the tree data structure.

         INPUT:
                nonodes          -  number of nodes including notes which are no
                                    longer valid
                tree             -  tree data structure with valid and
                                    invalid nodes
         OUTPUT:
                nonodes          -  actual numer of valid nodes
                tree             -  tree structure with only valid nodes
}
var     i,j,k: integer;
        actnonodes:tmaxnodes;    {actual number of nodes}
        ptr:targ;
        qtr:tres;
        labmap: array[tmaxnodes] of tmaxnodes;  {array to map old node index and

begin
  j:=0;
  for i:=1 to nonodes do
    if tree[i].funct[1] <> 'X' then begin
       {nodes with funct[1] = 'X' are no longer valid nodes}
      j:=j+1;
      labmap[i]:=j;
  end;
  actnonodes:=j;
  j:=0;
  for i:=1 to nonodes do begin
    with tree[i] do begin
      if funct[1]<>'X' then begin
```

```
          j:=j+1;
          tree[j].funct:=funct;
          tree[j].proctime:=proctime;
          tree[j].narg:=narg;
          tree[j].arg:=arg;
          ptr:=arg;
          if narg>0 then
            for k:=1 to narg do begin
              ptr^.no:=labmap[ptr^.no];
              ptr:=ptr^.next;
          end;
          tree[j].nres:=nres;
          tree[j].res:=res;
          qtr:=res;
          if nres>0 then
            for k:=1 to nres do begin
              qtr^.no:=labmap[qtr^.no];
              qtr:=qtr^.next;
          end;
        end;
      end;
    end;
    entrynode:=labmap[entrynode];
    nonodes:=actnonodes;
  end; {remnodes}

  {*****************************************************************************
                                                          DATAOUT
   ***************************************************************************** }
  procedure dataout(tree:typetree;nonodes:tmaxnodes;newnonodes:tmaxnodes);
  {

          Procedure to output the program graph to  a file set to text var out.

          INPUT:
                 nonodes          - total no. of nodes in the graph
                 tree             - graph with nodes in an array
          OUTPUT:
                 out              - output in file eq. to variable out
  }
  var     i,j: integer;
          ptr:targ;
          qtr:tres;

  begin
    writeln(out,'NO OF NODES',newnonodes);
    for i:=1 to nonodes do
      if tree[i].funct[1] <> 'X' then {if node is valid}
      with tree[i] do begin
        writeln(out,'node ',i);
        funct[1]:=' ';funct[2]:=' ';
        writeln(out,'funct              ',funct);
        writeln(out,'proctime                ',proctime:10:3);
        writeln(out,'narg               ',narg);
        ptr:=arg;
        if narg >0 then begin
          for j:=1 to narg do begin
            write(out,ptr^.no) ;
            if ptr^.dir='b' then write(out,'b');
            ptr:=ptr^.next;
          end;
          writeln(out);
        end;
        writeln(out,'nres               ',nres);
```

```
        qtr:=res;
        if nres > 0 then begin
          for j:=1 to nres do begin
            write(out,qtr^.no) ;
            if qtr^.dir='b' then write(out,'b');
            qtr:=qtr^.next;
          end;
          writeln(out);
        end;
        qtr:=res;
        if nres > 0 then begin
          for j:=1 to nres do begin
            write(out,qtr^.commtime:10:3) ;
            qtr:=qtr^.next;
          end;
          writeln(out);
        end;
        qtr:=res;
        if nres > 0 then begin
          for j:=1 to nres do begin
            write(out,qtr^.bustime:10:3) ;
            qtr:=qtr^.next;
          end;
          writeln(out);
        end;
        writeln(out);
      end;
end;    {dataout}


{*****************************************************************************
                                                        PARAMETERIZE
 ***************************************************************************** }
procedure parameterize(var tree:typetree;nonodes:tmaxnodes);
{

        Procedure to parameterize the processing time and communication times
        in the program graph. Procedure asks if parameterization is required
        and if so requests for the parameters. If parameterization option is
        used then the times in the graph are overruled. If however we only
        want to parameterize the communication times then if we assign a
        negative parameter to the processing time then the processing times for
        the nodes will be taken from the input graph data


}
var     i,j: integer;
        tmpchar:char;
        pr,buscomm,loccomm:real;
        qtr:tres;


begin
  write ('Want to parameterize Communication  and Processing times (y or n) ?');
  readln(tmpchar);
  if ((tmpchar = 'y') or (tmpchar='Y')) then begin
    write ('Processing time:'); readln(pr);
    write ('Local Communication time:'); readln(loccomm);
    write ('Bus Communication time:'); readln(buscomm);
    writeln('Pr.time = ',pr);
    writeln('LocComm.time = ',loccomm);{All arcs are given this local
                                                                comm. time}
    writeln('BusComm.time = ',buscomm);{All arcs are given this bus comm. time }
    for i:=1 to nonodes do begin
```

```
        with tree[i] do begin
          if pr >= 0 then proctime:=pr; {All nodes are given this proc. time if it
                                         is positive else retain original proc. times}
          if nres>0 then begin
            qtr:=res;
            for j:=1 to nres do begin
              qtr^.commtime:=loccomm;
              qtr^.bustime:=buscomm;
              qtr:=qtr^.next;
            end;
          end;
        end;
      end;
    end;
end;{parameterize}


{*****************************************************************************
                                                          UPREDUC
 ***************************************************************************** }
procedure upreduc(var tree:typetree; index:tmaxnodes; var newnonodes:tmaxnodes);
{
        Starting from node index this recursive procedure checks if the
        condition for combining the argument nodes and this node is
        satisfied. If it is then the functions of the argument nodes
        are copied to the index node. The index node's arguments will
        now be the arguments of the arguments. The result field of the
        arguments of the arguments has to be modified to reflect new
        results. If due to reduction we encounter two arcs between a
        pair of nodes we sum the comm times and replace them by a single
        arc. Note no upward reduction of OR nodes.

        INPUT:
              index          - present node which is being analyzed
              tree           - graph
        OUTPUT:
              tree           - graph after upward reduction
        PROCEDURE:
              upreduc        - recursive
}
var     n,i,k,m,l: integer;
        cond: real;
        singres: boolean;
        maxtime,sumproctime,largres: real;
        tnarg: integer;
        prevptr,ptr,rtr,firstptr,tptr:targ;
        prevqtr,qtr:tres;


begin
  ptr:=tree[index].arg;
  with tree[index] do begin
    if ((narg>0) and (funct[1]<>'D')) then
    if funct[1] = 'X' then
    writeln ('ERROR in UPREDUC - reference to invalid (nonexistent) node') else
    if funct[2] = 'O' then      {no upward reduction of OR nodes}
      for i:=1 to narg do begin
        if ptr^.dir <> 'b' then upreduc(tree,ptr^.no,newnonodes);{Only OR nodes
                                                 can have backward arcs as argument}
        ptr:=ptr^.next;
      end
    else begin
         maxtime:=0; sumproctime:=0; ptr:=arg; singres:=true;
```

```
i:=1;k:=narg;
while ((ptr<>nil) and (singres=true)) do begin
  if tree[ptr^.no].nres >1 then begin
    qtr:=tree[ptr^.no].res;
    n:=0; largres:=0;
    m:=tree[ptr^.no].nres;
    for l:=1 to m do begin
      if qtr^.no = index then begin
        n:=n+1;
        if qtr^.commtime>largres then largres:=qtr^.commtime;
      end else singres:=false;
      qtr:=qtr^.next;
    end;
    if n=m then begin
      tree[ptr^.no].res^.commtime:=largres;
      tree[ptr^.no].res^.next:=nil;
      tree[ptr^.no].nres:=1;
      tptr:=ptr;tptr:=tptr^.next;prevptr:=ptr;
      while tptr<> nil do begin
        if tptr^.no = ptr^.no then begin
          prevptr^.next:=tptr^.next;
          narg:=narg-1;
        end else prevptr:=tptr;
        tptr:=tptr^.next;
      end;
    end;
  end;
  if singres=true then begin
    sumproctime:=sumproctime+tree[ptr^.no].proctime;
    if ( tree[ptr^.no].proctime +tree[ptr^.no].res^.commtime)> maxtime
    then
      maxtime:=tree[ptr^.no].proctime + tree[ptr^.no].res^.commtime;
  end;
  i:=i+1;
  ptr:=ptr^.next;
end;
ptr:=arg;
cond:=maxtime-sumproctime;          {compresion condition}
 {combination of node and its arguments}
if ((cond<=0) or (singres=false)) then          {no compresion}
  for i:=1 to narg do begin
    upreduc(tree,ptr^.no,newnonodes);
    ptr:=ptr^.next;
  end
else begin                                      {compresion}
  tnarg:=0;tptr:=arg;firstptr:=nil;
  m:=0;
  repeat m:=m+1 until funct[m]=' ';
  for i:=1 to narg do begin
    if tree[tptr^.no].narg > 0 then  begin    {new arg for index}
      tnarg:=tnarg+tree[tptr^.no].narg;
      if firstptr=nil then begin
        rtr:=tree[tptr^.no].arg;
        firstptr:=rtr;
      end else begin
        rtr^.next:=tree[tptr^.no].arg;
        rtr:=rtr^.next;
      end;
      for k:=1 to tree[tptr^.no].narg do begin  {res of arg of args
                                                          modified}
        qtr:=tree[rtr^.no].res;
          for l:=1 to tree[rtr^.no].nres do begin
            if qtr^.no=tptr^.no then qtr^.no:=index;
            qtr:=qtr^.next;
```

```
                    end;
                  if rtr^.next <> nil then rtr:=rtr^.next;
                end;
              end;
              k:=2;                          {copy functions of arg to index node}
              funct[m]:=',';
              repeat m:=m+1; k:=k+1; funct[m]:=tree[tptr^.no].funct[k]
              until tree[tptr^.no].funct[k]=' ';
              if tree[tptr^.no].funct[2]='0' then funct[2]:='0';
              if tptr^.no=entrynode then entrynode:=index;
              tree[tptr^.no].funct[1]:='X';    {Arg node no longer part of tree}
              newnonodes:=newnonodes-1;
              tree[tptr^.no].arg:=nil;
              tptr:=tptr^.next;
            end;
            arg:=firstptr;
            narg:=tnarg;                       {No. of arg is sum of narg of args}
            proctime:=proctime+sumproctime;    {new proc. time is sum of proc.
                                                  times of all the nodes combined}
            upreduc(tree,index,newnonodes);    {Try reduction with new arguments }
          end;
        end;
        if funct[1] <> 'X' then funct[1]:='D';          {Mark it as observed }
         {'D' indicates that upreduc has encountered this node}
      end;
end;    {upreduc}

{*****************************************************************************
 ****************************************************************************
                                                          DNREDUC
 ****************************************************************************** }
procedure dnreduc(var tree:typetree; index: tmaxnodes;var newnonodes:tmaxnodes);
{

        Starting from node index this recursive procedure checks if the
        condition for combining the result nodes and this node is
        satisfied. If it is then the functions of the result nodes
        are copied to the index node. The index node's results will
        now be the results of the result. The result field of the
        result of the result has to be modified to reflect new
        results. If due to reduction we encounter two arcs between a
        pair of nodes we sum the comm. times and replace them by a single
        arc.

        INPUT:
              index          -
              tree           -
        OUTPUT:
              tree           -
        PROCEDURE:
              dnreduc        - recursive
}
var     n,i,k,m,l: integer;
        cond: real;  {compression condition - compress if >0 }
        singarg: boolean;
        maxtime,sumproctime,largarg: real;
        tnres: integer;
        ptr,prevptr:targ;
        qtr,rtr,firstqtr,tqtr,prevqtr:tres;


begin
  qtr:=tree[index].res;
  with tree[index] do begin
    if ((nres>0) and (funct[1]<>'U')) then
```

```
if funct[1] = 'X' then
writeln ('ERROR  in DNREDUC - invalid node encountered') else
if ((funct[2] = 'S') or (funct[3] = 'S')) then{no down reduction for SWITCH}
  for i:=1 to nres do begin
    if qtr^.dir <> 'b' then dnreduc(tree,qtr^.no,newnonodes);
    qtr:=qtr^.next;
  end
else begin
      maxtime:=0; sumproctime:=0; qtr:=res; singarg:=true;
      i:=1;k:=nres;

       {singarg will be true if the result nodes of index node have
        only one argument which is the index node or all its arguments
        are the index node. Even though we don't admit two arcs in the same
        direction between the same pair of nodes initially, this can occur
        after combinations}
      while ((qtr<>nil) and (singarg=true)) do begin
        if tree[qtr^.no].narg >1 then begin
           {if the result has more than one argument}
          ptr:=tree[qtr^.no].arg;
          n:=0;largarg:=0;
          m:=tree[qtr^.no].narg;
          for l:=1 to m do begin
            if ptr^.no= index then begin
              n:=n+1;
            end else singarg:=false;
            ptr:=ptr^.next;
          end;
          if n=m then begin
            {if all the arguments of the result node are the index node,
             i.e., the node under consideration}
            {if parallel arcs from index to result then replace by a single
             arc}

            tree[qtr^.no].narg:=1;
            tree[qtr^.no].arg^.next:=nil;
            tqtr:=qtr;tqtr:=tqtr^.next;prevqtr:=qtr;
            largarg:=0;
            while tqtr<>nil do begin
              if tqtr^.no=qtr^.no then begin
                prevqtr^.next:=tqtr^.next;
                nres:=nres-1;
                if tqtr^.commtime>largarg then largarg:=tqtr^.commtime;
              end else prevqtr:=tqtr;
              tqtr:=tqtr^.next;
            end;
            qtr^.commtime:=largarg;
          end;
        end;
        if singarg=true then begin
          sumproctime:=sumproctime+tree[qtr^.no].proctime;
          if ( tree[qtr^.no].proctime +qtr^.commtime)> maxtime then
            maxtime:=tree[qtr^.no].proctime + qtr^.commtime;
        end;
        i:=i+1;
        qtr:=qtr^.next;
      end;
      qtr:=res;
      cond:=maxtime-sumproctime;          {compresion condition}
      if ((cond<=0) or (singarg=false)) then          {no compresion}
        for i:=1 to nres do begin
          dnreduc(tree,qtr^.no,newnonodes);
          qtr:=qtr^.next;
        end
```