APPENDIX A

MultiPascal Language
MultiPascal is a language with a simple set of high level parallel programming abstractions that have sufficient power to represent parallel algorithms for multiprocessors and multicomputers. It is an extension of Pascal language, with an addition of features for creation and interaction of parallel processes. MultiPascal is designed to be machine independent and can run on a wide variety of parallel computers, including multiprocessors with shared memory and multicomputers based on message passing between processors with local memory.

As the MultiPascal system executes a program, it keeps track of the relative timing of all the processes and generates a range of performance statistics to help the user understand the behavior of the program. The program is compiled into a pseudo-machine code, which is interpreted rather than directly executed. As the MultiPascal interprets the program, it uses an estimated execution time for each pseudo code instruction and keeps a running total of the execution time of the program. The time charged for each pseudo code instruction varies from 1 to 6 time units, depending upon the relative complexity of the instruction.

### A.1 MultiPascal Constructs

On a uniprocessor system, MultiPascal simulates hardware processors as processes and shared memory as shared variables. Some of the main MultiPascal constructs are as follows:

1. **FORALL statement**

   FORALL is the most powerful method of creating parallel processes in MultiPascal; a parallel form of the FOR loop in which all the loop iterations are executed in parallel rather than sequentially. As an
example, consider the following parallel program for calculating the square root of 100 elements of an array.

```pascal
PROGRAM ParallelSquareroot;
VAR A: ARRAY[1..100] OF REAL;
i : INTEGER;
BEGIN
  . . .
  FORALL i := 1 TO 100 DO
    A[i] := SQRT(A[i]);
  . . .
END.
```

The above FORALL statement creates 100 copies of the enclosed assignment statement and makes each one a separate parallel process with its own unique value of the index variable i. Each of these 100 processes may be executed on a different processor, all in parallel. In the FORALL statement, the keyword GROUPING may be used to group together a certain number of index values in each process.

The general syntax of the FORALL statement is as follows:

```
FORALL <index-variable> := <initial> TO <final> [GROUPING <size>] DO <statement> ;
```

where <initial>, <final>, and <size> may be any integer-valued expressions. The set braces "{}" enclose optimal entities. If GROUPING is omitted, then the default group <size> does not evenly divide the number of index values in the specified range, then the last child process will have less than <size> index values. FORALL statements just like sequential FOR statements can be nested also.
The following sequence of action results from the FORALL statement which has \( i \) as the initial value \( n \) as the final value, and no grouping:

- Create the \( n \) child processes from <statement>
- Wait until all the \( n \) child processes have terminated
- Continue execution after the FORALL

2. The FORK Operator

FORK operator turns an individual statement into a child process. By preceding any statement with the FORK operator, it becomes a child process running in parallel with its parent. The general syntax of FORK is as follows:

\[
\text{FORK } <\text{statement}> 
\]

The statement can be any valid MultiPascal statement. This statement will become a child process that is executed on a different processor. The parent will continue the execution without waiting for the child in any way. Consider the following example:

\[
\text{FORK FOR } i := 1 \text{ to } 10 \text{ DO } A[i] := i; \\
z := \text{SQRT}(x)
\]

The parent process creates a child process consisting the FOR statement. While the child process is still running, the parent will execute the assignment to variable \( z \) and then continue to run in parallel with its child.

Although the parent process does continue with its execution while its FORK children are still running, the parent is not permitted to terminate until all its children have terminated. If the parent reaches the program code while one or more of its child process is still running, the
execution of the parent will be suspended until all the children terminate, and then the parent will also terminate.

3. The JOIN Statement

JOIN executed by the parent forces it to wait for the child to terminate. If the child has already terminated, then the execution of JOIN will have no effect on the parent. For example consider the following portion of a parent process:

```
FORK Normalize(A);
FOR i := 1 TO 10 DO
  B[i] := 0;
JOIN;
```

The parent creates a FORK child consisting of a call to procedure Normalize. Then while the child is executing, the parent will execute the following FOR loop to initialize the array B. After the loop is finished, the parent executes the JOIN statement. If the FORK child has already terminated, the parent just continues execution after the JOIN immediately. However, if the child is still running, the parent will suspend execution at the JOIN until the child terminates, then continue following the JOIN.

If the parent has multiple FORK children, then the termination of any of these children will satisfy any JOIN in the parent. Multiple JOIN statements can be executed to wait for them all to terminate. The execution of each JOIN by the parent will match one single FORK child termination.

```
FOR i := 1 TO 10 DO
  FORK Compute(A[i]) ;
FOR i := 1 TO 10 DO
```
JOIN;

If the parent mistakenly executes more JOIN statements than it has children, the parent will be suspended forever, resulting in an execution "deadlock" in the program.

4. The LOCK and UNLOCK operators

In MultiPascal, SPINLOCK is a built-in data type. In the declaration section of the main program or any procedure, any variable name may be declared as having type SPINLOCK. Once a variable is declared in this way, it can be used in LOCK or UNLOCK operations in the program. MultiPascal restricts the use of LOCK and UNLOCK to spinlock variables. Attempting to ally these operations to variables of other types will result in compiler error. LOCK and UNLOCK are always used as pairs, surrounding a few statements of the program. The compiler will allow these two operations to appear anywhere in the program, provided that they are applied to valid spinlock variables. Consider the example:

PROGRAM lockcheck;
VAR n: INTEGER;
  l: SPINLOCK;
BEGIN
  ...
  LOCK (l); (* Enter and lock the spinlock *)
n := n + 1;
  UNLOCK (l); (* Unlock the spinlock to allow another to enter *)
  ...

The above code guarantees, that only one process at a time can update variable n.
A.2 Simulation of Parallel Architectures

When the MultiPascal simulation system is used, the default architecture is a shared multiprocessor with a maximum of 512 processors available (256 in the MS-DOS version). The programmer can, however, override this default architecture and specify a wide range of other architectures.

Example:

```
PROGRAM Sample ;
ARCHITECTURE SHARED (64) ;
```

The above example chooses a shared memory architecture with 64 processors. ARCHITECTURE specification must immediately follow the PROGRAM line. The other architecture specifications supported by MultiPascal are:

- **SHARED(p)**
  - p - total number of processors available in shared architecture

- **LINE(n)**
  - n - total number of processors in this Line multicomputer topology

- **RING(n)**
  - Same as LINE, except that the last processor in the line is connected back to the processor 0.

- **MESH2(m)**
  - 2D mesh topology with m rows and n columns

- **TORUS(m)**
  - Identical to 2D mesh topology, except that the processors at each boundary have direct connections to the corresponding processors on the opposite boundary

- **MESH3(m)**
  - 3D mesh topology. It will have a total of $m^3$ processors.
HYPERCUBE\( (d) \) Hypercube topology with dimension specified by the parameter \( d \). Thus the total number of processors is \( 2^d \).

FULLCONNECT\( (p) \) Fully connected multicomputer topology, in which each processor has a direct physical communication link to every other processor.

A.3 Interactive MultiPascal Commands

The MultiPascal interactive system contains a variety of commands to help the programmer isolate bugs and performance bottlenecks. There are also several features in the language to monitor the program performance. The system simulates the actual timing behavior of the program as it would run on the target multiprocessor hardware. The programmer can view various performance statistics such as elapsed execution time, parallel speedup factor, and processor utilization. A detailed visual profile of processor utilization may also be generated to help programmer understand the dynamics of program performance.

Following is a summary of available commands in the interactive MultiPascal system.

- **RUN** Initialize and run the program from the beginning
- **EXIT** Terminate the MultiPascal system
- **LIST n:m** List program source lines \( n \) through \( m \)
- **BREAK n** Sets a breakpoint at program line \( n \)
- **CLEAR BREAK n** Clears the breakpoint from line \( n \)
- **CONT** Continue the execution after the breakpoint
- **STATUS p:q** Displays status of processes \( p \) through \( q \)
- **STEP n** Continues execution for \( n \) lines in current STEP
**STEP PROCESS** p
Sets current STEP process number to p

**WRITE** p name
Writes out value of variable “name” in process p

**TRACE** p name
Makes variable “name” a trace variable

**DISPLAY**
Displays list of breakpoints, trace variables and alarms

**CLEAR TRACE** m
Clears the trace from memory location m

**TIME**
Gives elapsed time since start of program and since last break

**UTILIZATION** p:q
Gives utilization percentage for processors p to q

**PROFILE** p:q t
Causes utilization profile to be generated for processes p to q every t time units

**PROFILE OFF**
Turns off profile

**ALARM** t
Sets an alarm to go off after t time units from the beginning of the program execution. Program execution is suspended when the alarm goes off.

**ALARM OFF**
Disables alarm from going off in future

**VARIATION ON (OFF)**
Creates randomly chosen variations in the speed of each processor, to help determine the whether the program has timing dependent bugs.

**CONGESTION ON (OFF)**
For multicomputer architecture. Determines the communication time for messages

**DELAY** d
For multicomputer architectures. Sets the basic communication delay between processors with a direct communication link.