PARSEC User Manual
For PARSEC Release 1.1
Revised in September 1999

Manual written by Richard A. Meyer and Rajive Bagrodia

PARSEC was developed by members of the UCLA Parallel Computing Laboratory
Mineo Takai, Jay Martin, Richard Meyer, Brian Park, Ha Yoon Song

http://pcl.cs.ucla.edu/
parsec@cs.ucla.edu

For more information on the PARSEC simulation language and related projects, please visit the Parallel Computing Laboratory web page at: http://pcl.cs.ucla.edu/.

1 Introduction

PARSEC (for PARallel Simulation Environment for Complex systems) is a C-based discrete-event simulation language. It adopts the process interaction approach to discrete-event simulation. An object (also referred to as a physical process) or set of objects in the physical system is represented by a logical process. Interactions among physical processes (events) are modeled by timestamped message exchanges among the corresponding logical processes.

One of the important distinguishing features of PARSEC is its ability to execute a discrete-event simulation model using several different asynchronous parallel simulation protocols on a variety of parallel architectures. PARSEC is designed to cleanly separate the description of a simulation model from the underlying simulation protocol, sequential or parallel, used to execute it. Thus, with few modifications, a PARSEC program may be executed using the traditional sequential (Global Event List) simulation protocol or one of many parallel optimistic or conservative protocols.

In addition, PARSEC provides powerful message receiving constructs that result in shorter and more natural simulation programs. Useful debugging facilities are available.

The PARSEC language is derived from Maisie, but with several improvements, both in the syntax of the language and in its execution environment. Appendix “C” contains information about converting existing Maisie programs into PARSEC.

1.1 Organization

The next section describes the PARSEC language, defining especially the two main enhancements that PARSEC makes to C -- constructs for defining and creating (simulation) entities, and constructs for message communication. Section 3 contains information about the PARSEC programming environment, including instructions for using the compiler, and debugging support for PARSEC. Section 4 lists the additional requirements for running PARSEC simulations in parallel. Appendix A includes a list of all the PARSEC library functions which are available to the programmer. **Appendix C contains important information about upgrading existing programs to use the new system.** Appendix D includes information about limitations and common problems in executing PARSEC programs. Appendix E contains the latest information on how to get and install PARSEC.

1.2 Notation

The reader is assumed to be familiar with the C programming language, according to the ANSI standard. The syntax of PARSEC statements is given in BNF using the following symbols and conventions:

Terminals	:	terminal symbols are given in boldface style
Nonterminals	:	nonterminal symbols are in italic style
ident	:	a C identifier (see Appendix 2.2 for identifiers reserved by PARSEC)
[]	:	symbols occuring within brackets are optional
[]…	:	symbols may be repeated 0 or more times
|	:	alternative.

2 The PARSEC Language

The PARSEC language is based on C, but introduces several new features. PARSEC programs consist of **entities**, which exchange **messages**. The following sections introduce these two features and describe how to construct a program from them.

2.1 Types and Keywords

PARSEC defines the following types and keywords, which are described in much greater detail subsequently:

clocktype : an abstract numeric type for representing time
ename : an entity identifier
message : a structure defining the contents of a message

PARSEC Reserved Words					
after	**at**	**clocktype**	**driver**	**ename**	**entity**
finalize	**in**	**message**	**new**	**or**	**receive**
self	**send**	**stacksize**	**timeout**	**to**	**when**

Appendix A contains a comprehensive list of built-in functions which are available to the user. These functions may also lead to name conflicts, and PARSEC also reserves all function and variable names prefixed with “MC_” or “pc_” for its internal functions.

2.2 Entities

A PARSEC program is a collection of C functions and entity definitions. An entity definition (or an entity type) describes a class of objects. Instances of an entity type may be created to model object(s) in the physical system. For instance, an entity type *Server* may be defined to model server objects; specific instances of the *Server* entity type are created to model server objects in the physical system. Henceforth we will use the term entity to mean an entity instance.

2.2.1 Entity Definition

The definition of an entity type is similar to the definition of a C function. The syntax is as follows:

```
entity-def    ::=  entity ident ([parameters]) [stacksize (size_expr)] body
parameters    ::=  A C parameter list
size_expr     ::=  A unary integer expression  (see Remarks below)
body          ::=  {a series of PARSEC and C statements [finalize_st]}
finalize_st   ::=  finalize { a series of C statements }
```

The entity heading is similar to an ANSI C function heading; it specifies a name for the entity type and gives a list of typed parameters. The entity body is a compound statement, possibly terminated with a **finalize** block, that describes the actions executed by an entity type. The **finalize** block includes a set of C statements describing what to do when an entity terminates. It is described further in Section 2.2.3. The first example below gives the heading for an entity type *Manager* with one integer parameter *max_printers*. The second example shows an entity type *Sort* that includes a **finalize** statement and declares two formal parameters: an integer *n* and an array *a* with a fixed size of 20.

Examples:

```
entity  Manager (int max_printers ) {
        int units;
        . . .
}

entity  Sort (int n, int a[20]) {
        . . .

        finalize {
                . . .
        }
}
```

Remarks:

1. An array parameter must be declared with an explicit size because arrays are passed *by value* in PARSEC. (This is because the creating entity may be located on a different processor.) I *.e.* if *x* is an entity parameter, ``int *x*[];" is illegal as its type declaration. Furthermore, the array size must be a constant expression.
2. *static* variables declared in an entity body are no different from other local variables because each entity-instance maintains a different copy of the variable. This is different from a function definition in which different *calls* to the function share the same copy of the *static* variable.
3. The **stacksize** keyword is used in the following way. Each entity instance is executed in a semi-private address space with its own stack pointer. This stack must be large enough to handle recursive function calls and the local variables of the entity. The default size is 200KB. Please see Appendix D.1 for more information about using the **stacksize** specifier.
4. Entities must be declared before they are used, in much the same way that C functions must be declared.

2.2.2 Entity Creation

PARSEC supports dynamic and recursive entity creation. A PARSEC entity is created by using the **new** statement, which has the following syntax:

```
new-st        ::=  [ename-expr  = ] new ident([arg]...) [at node-no]
ename-expr    ::=  an expression of type ename
arg           ::=  C argument expression
node-no       ::=  a positive integer-valued expression
```

The following example creates a new instance of the *Manager* entity type (defined in the previous section) and saves the unique identifier assigned to the entity in variable *m1*. An entity may refer to its own identifier by using keyword **self**.

Example:

```
{
        ename m1;
        ...
```

```
        ml = new Manager (10);
        ...
}
```

By default, the new entity executes on the same processor as its creator entity. The **new** statement may optionally include an **at** clause to specify a different processor (node) for execution of the new entity. The *node-no* in the **new** statement may be an arbitrary positive integer; it is mapped to a specific node on the parallel architecture as follows: the actual number of nodes, say *N*, used in an execution of the program is specified as a command line argument (see Section 3). The *node-no* in a **new** statement is interpreted as *node-no modulo N* to determine the target processor. Although entity placement is automatically done modulo the number of nodes, the function *pc_num_nodes()* is provided which returns the number of nodes being used for a particular execution. The **at** clause is ignored for sequential implementations of the program.

Example:

```
{
        ename c[10];
        ...
        for (i = 0; i < 10; I++)
                c[i] new Client () at  I;
                ...
}
```

Execution of a **new** statement returns a unique identifier of type **ename**. This is a new type introduced by PARSEC; variables of this type are used only to refer to entities. Very few operations are defined on variables of type ename: they may be passed as entity or message parameters and may be assigned a value by a **new** statement. In addition, a function **ename_cmp** (*e*1,*e*2) is defined by the runtime library, where *e*1 and *e*2 are of type **ename**. The function returns a *non-zero* value if and only if the two variables are identical.

As mentioned before, an entity may include array parameters. However, unlike C functions, the array parameters are passed by value. In the following example, the *driver* entity demonstrates how to instantiate entity *Sort* (defined in the previous section).

Example:

```
entity driver () {
        ename s1;
        int x[20];

        s1 = new Sort (20, x);
        ...
}
```

2.2.3 Entity Termination

PARSEC entity may terminate itself in any of the following ways: by executing a C return statement, by “falling off the end” of the entity-body. All entities still active at the end of a simulation will be terminated by the runtime system.

If an entity definition includes a **finalize** statement, the body of the finalize statement will be executed upon a normal termination of each instance of that entity type. The finalize statement is most useful for collecting the results of a program at its conclusion.

2.2.4 Friend Functions

Friend functions are provided for backward compatibility with Maisie. However, we have discovered from experience that programs with friend functions are difficult to debug and less efficient than ordinary functions; we recommend that you use them sparingly or not at all. Local data of an entity is inaccessible by any function that is called by the entity. It is, however, possible to allow a function to access local variables of the calling entity, without explicitly passing the variable as a parameter to the function by declaring friend functions. (Friend functions are very similar to C++ class methods.) In both its declaration, which must precede the associated entity's definition, and in its definition, the function name must be prefixed with the entity-name, and the function definition must *follow* the entity definition in the same file. Friend functions must be compiled with the **-** ff command flag. The following example illustrates the use of friend functions.

Example:

```
entity Sort (int, int);
void Sort::sorting(void);

entity Sort (int n, int a[20]) {
       ...
       sorting();
}

/* the friend function can access the parameter a defined in the Sort entity */
void Sort::sorting() {
       int i, t;
       for(t = a[0], i = 1; i < n; i++) {
              if(a[i t],t)
                     ...
       }
}
```

2.3 Message Communication

Entities communicate with each other via buffered message passing. A unique message buffer is associated with each entity. Asynchronous send and receive primitives are provided to respectively deposit and remove messages from the message buffer of an entity. The **receive** primitive may also be used to advance the simulation clock.

PARSEC uses typed messages. A message-type consists of a name and a parameter list. The following syntax is used to define a message-type:

```
message-def     ::=  message ident { declarations} [ident]...;
declarations    ::=  [type ident[,ident]...;]...
type            ::=  ename | clocktype | message ident|any C type declaration
```

Message definition is syntactically similar to the declaration of a C *struct*. Message parameters may be viewed as fields defined within a struct and are referenced using the same '.' operator as used to reference fields within a *struct*. A message-type with an empty parameter-list is used to define signals (*e.g.*, acknowledgments). Message type declarations are treated as global (even if declared within an entity body), so it is standard to declare all message types at the head of the file, or in a separate header file.

As with entity parameters, message parameters can include arrays, but they must be given explicit sizes, and they will be passed *by value*.

It is also possible to use pointers as message parameters, *but it is very dangerous, and not recommended at all!* On a distributed memory parallel architecture, the pointer values will be meaningless on remote processors. Even on a shared memory architecture or a uni-processor, using pointers correctly requires expert care. Otherwise, their use may lead to incorrect results, because messages at different entities might not be accepted in the order you expect.

In the following example, we elaborate the entity type *Manager* to include definition of two message types: *Release* with an integer parameter, and *Request* with two parameters: *id* of type ename and an integer parameter called *units*. The local variable *oldrequest* is declared to store messages of type *Request*.

Example:

```
message Done {};
message Release {
        int units;
};
message Request {
        int   units;
        ename id;
};

entity  Manager (int max_printers) {
        int units;
        message Request oldrequest;
        ...
}
```

Remarks:

1. Message types must be globally unique within a program, and should be declared globally. Using the same identifier for two different message types (i.e. with a different set of parameters) in the same program will be treated as an error.
2. For parallel programming, one should avoid using pointers as parameters in message types. While such declarations are legal, pointer values will have no meaning when sent to another processor.

2.3.1 Sending Messages

An entity sends a message to another by using a **send** statement. This statement has the following syntax:

```
send-st         ::=  send msg-expr  to  ename-expr [after time-expr ];
ename-expr      ::=  an expression of type ename
msg-expr        ::=  msg-type {  [arg]... } | msg-ident
time-expr       ::=  an integer expression whose value is   > 0
arg             ::=  array-param | C expression
array-param     ::=  a pointer expression [:: a positive integer-valued
                     expression]
msg-type        ::=  a message type defined for entity ename-expr
msg-ident       ::=  a variable of type msg-type
```

The **send** statement performs an asynchronous send: the sending entity copies the message parameters into a memory block, delivers the message to the underlying communication network, and resumes execution. Every message is implicitly timestamped with the current value of the simulation clock. The programmer may specify a different timestamp by using the optional ``**after** *time-expr*'' attribute--this causes the timestamp of the message to be set to the current simulation time **plus** *time-expr*. (In a simulation, the **after** clause is used to model transmission delay, i.e. the message leaves here *now*, but arrives there after *time-expr* time units.) A message is delivered to the destination buffer at the simulation time specified by its timestamp.

The following examples demonstrate several ways of sending a *Request* message to entity *s1*, an instance of the entity type *Manager* defined previously. The first statement specifies the message parameters explicitly; the second specifies that the message be copied from variable *oldrequest*. The last example will cause the Request message to be delivered to the message buffer of the destination entity five time units from the simulation time of the sending entity.

Example:

```
message Request oldrequest; /* initialize oldrequest */
send Request{10, self} to s1;
send oldrequest to s1;
send oldrequest to s1 after 5;
```

Remarks:

1. Messages are received by an entity in their timestamp order. Messages of a single type with the same timestamp from a common source are received in the order they are sent; however no a priori ordering can be assumed for messages with the same timestamp received from multiple sources.

Array Parameters: A message may include array parameters that are passed *by value*. If a formal parameter of a message is declared as an array, the corresponding actual parameter will be treated as an array. The actual parameter may specify an array slice by specifying the position of the first element followed by an optional size that specifies the number of bytes to be transmitted. When specifying the size, you generally specify the number of elements multiplied by the size of an element, as in the following example. If the size is omitted, it will default to the size declared for the corresponding formal parameter. The following example passes a message value with elements x[10]..x[14] to entity *s1*.

Example:

```
message value { int count; int a[10]; };
In sending entity:
int x[20];
...
/* Send 5 integers from x[10]..x[14] */
send value { 5, &x[10]::5*sizeof(int) } to s1;
...
```

Remarks:

1. If the formal array parameter has size N, then by default, N contiguous pieces of data will be copied from the actual parameter.
2. If the actual parameter is larger than the formal parameter, only a portion the size of the formal parameter will be copied into the message.
3. If the actual parameter is smaller than the formal parameter, the user must specify a slice the size of the actual parameter. Otherwise, there is the possibility of a crash when the system copies past the end of the array.

4. If the user specifies a slice larger than the size of the formal parameter, it will be reported as an error.

2.3.2 Receiving a Message

An entity accepts messages from its message-buffer by executing a **receive** statement, which has the following syntax:

```
receive-st        ::=  resume-clause [or  timeout-resume]
resume-clause     ::=  receive  (msg-list)  [when  (guard)] statement
timeout-resume    ::=  receive-st | timeout-st
timeout-st        ::=  timeout  after | in (delay-time) statement
msg-list          ::=  msg-type mvar [, msg-list]
delay-time        ::=  a C expression of type clocktype
guard             ::=  a C expression without side-effects
mvar              ::=  a variable of type msg-type
statement         ::=  Any C or PARSEC statement
```

The receive statement consists of one or more resume clauses, and possibly a timeout clause. Each resume clause consists of a read-only message variable, and an optional *guard* followed by a statement. If the message buffer contains exactly one enabling message, the message is removed from the buffer and delivered to the entity in variable *mvar*. The variable *mvar* is a read-only variable, and its scope extends only to the statement that is part of the resume clause. The timeout clause specifies a wait time (t_c), and may specify either *timeout-first* or *timeout-last* semantics, using the **in** and **after** keywords, respectively. Timeout semantics will be discussed subsequently.

When a message is received, the internal clock of the entity will advance to the greater of 1) the time specified on the timestamp of the message, or 2) the current time of the entity. Thus, the entity's clock moves monotonically forward.

The following code segment shows two simple examples. The first is a simple receive statement with a single clause. The entity will block until a message of type Request arrives. That message is then copied into the *req* variable. The second statement is more complex, having two resume clauses and a timeout clause, only one of which will be executed. If either a Request or a Release message arrives before 5 time units, it will be accepted. If either arrives, the one with the earliest time stamp will be accepted. If they have the same time stamp, one will be chosen non-deterministically. If neither a Request nor Release message arrives before 5 time units have passed, the timeout clause will be executed.

Example:

```
message Request req;
int units;

/* simple receive */
receive (Request r) {
       req = r;
}

/* complex receive */
receive (Request r) {
       req = r;
}
or receive (Release r) {
       units = r.units;
}
or timeout in (5) {
       ...
```

```
}
```

Note that when the Request message is received, the entire message is copied into a local variable, but when the Release message is received, only the value of *units* is copied. Both usages are acceptable.

Remarks:

1. Receive statements can occur in functions as well as in entities.
2. Each message type can appear in only one clause of a receive statement.

2.3.3 Guards

The guard is a side-effect free expression. If omitted, the guard is assumed to be the boolean constant *true*. A guard is said to be *local* if it can be evaluated using only entity variables.

A resume clause with message type m_t and guard b_i is said to be *enabled* if the message buffer contains a message of type m_t and b_i evaluates to *true.* (b_i is evaluated only if the buffer contains a message of type $m_{t.}$) The corresponding message is called an *enabling message*. In general, the buffer may contain many enabling messages.

Example:

```
receive (Request r) when (r.units < 5) {
       req = r;
}
or receive (Release r) {
       units = r.units;
}
```

The following example illustrates simple receive statements with a single resume clause. The first receive statement is enabled if the message-buffer contains a *Request* message (as defined in entity *Manager* at the beginning of this section). The second receive statement is enabled only if the buffer contains a *Request* message whose parameter *units* is not greater than the entity variable *units.*

```
receive (Request req) { ...}
receive (Request req) when (req.units <= units) {...}     /* First-Fit */
```

If two or more resume clauses in a receive statement are enabled, the timestamps on the corresponding enabling messages are compared and the message with the earliest timestamp is removed and delivered to the entity. If the message timestamps are equal, and neither one of the messages is a timeout, an enabling message is selected non-deterministically for delivery to the entity (timeouts are discussed subsequently). Consider the Manager entity type in Figure 1. The entity receives messages of type *Request* and *Release*, and sends a message of type *Done*, all of which were defined earlier. Both resume conditions in the receive statement of the entity will be enabled if the buffer contains a *Release* message and also a *Request* message that satisfies the corresponding guard.

An entity is said to be *suspended* if none of its resume clauses are enabled. A suspended entity resumes execution if it receives an enabling message. If an entity executes a receive statement that includes only local guards and all guards evaluate to *false*, the runtime system will print a warning message, and skip the corresponding receive statement.

2.3.4 Timeout

If a receive statement executed by an entity includes a timeout clause with wait time t_c, execution of the statement schedules a timeout for the entity. PARSEC supports both *timeout-first* and *timeout-last*

```
entity Manager(int max_printers) {
    int units = max_printers;
    for (;;)
        receive (Request req) when (req.units <= units) {
            units -= reg.units;
            send Done{} to req.id;
        }
        or receive (Release rel) {
            units += rel.units;
        }
}
```

Figure 1: A Resource Manager

semantics through the use of the **in** and **after** keywords. These are two different ways of handling simultaneous events. Timeout-first is much more efficient, and is recommended in almost all cases, while timeout-last is more powerful, but cannot always be guaranteed. In the following two examples, assume that the statements are issued at time 0.

Example 1:

```
receive (Ack ack) {
    /* process acknowledgement... */
}
or timeout in (10) {
    /* resend the message */
}
```

Example 2:

```
receive (Ack ack) {
    /* process acknowledgement... */
}
or timeout after (10) {
    /* resend the message */
}
```

In example 1, the system will timeout in 10 time units (i.e., at time 10), *before* any Ack messages with timestamp 10 are accepted. In example 2, the system will timeout *after* any Ack messages with timestamp 10 are accepted. Another way to look at it is that in example 1, the system will timeout even if there is an Ack message with timestamp 10, but example 2 will timeout only if there are no Ack messages with timestamp 10.

From a modeling perspective, both semantics can be useful. For example, timeout-first semantics can be used to model a wireless radio transmission - if no interference is detected during an interval, the message is successful, messages arriving at the end of the interval do not interfere. Timeout-last semantics can be used to collect a set of simultaneous events and execute them together.

2.3.5 Hold Statement

In order to suspend an entity unconditionally for a specified duration to simulate activities like servicing a request, the **hold** statement has been introduced. This statement has the following syntax:

```
hold-st          ::=   hold(delay-time);
delay-time       ::=   a C expression of type clocktype
```

For example:

hold(5);

The statement **hold**(*delay-time*); will advance the simulation clock of the entity by *delay-time* time units.

2.3.6 Non-blocking Receive

When an entity executes a receive statement, it may be suspended if its buffer does not contain an enabling message. If the delay-time in the timeout clause of a receive statement is specified to be 0, the entity will resume execution at the current simulation time. In the following example, if the message buffer of the entity does not contain any *Request* message, a timeout message will be delivered to the entity with the current simulation time. This ensures that the entity will not be blocked indefinitely. Notice that the example uses the **timeout after** form of the timeout clause. This is the only case where timeout-last semantics are absolutely necessary.

Example:

```
receive (Request r) {
        process-this-message;
}
or timeout after (0) {                /* Request message unavailable */
        do-something-else;
}
```

2.3.7 Advanced Message Receive Constructs

Qhead(): In a resume clause with message type m_t and guard b_i, if b_i\ references message parameters (as in the *Request* message in Figure 1), the corresponding resume clause is enabled if any message of type m_t in the buffer satisfies the guard. In particular if the first message in the buffer requests a large number of units, it is possible that it may never be satisfied while smaller subsequent requests are serviced continuously. The first receive statement in the following fragment illustrates this situation. Alternately, it is sometimes desirable to define the resume condition such that it is enabled only if the first message of type m_t in the buffer is an enabling message, and is disabled otherwise. Among other things, this may be used to prevent the kind of starvation scenarios outlined in the preceding situation. PARSEC provides a function called **qhead**(m_t) where parameter m_t is a message type. The function returns a copy of the first message of type m_t in the message buffer; if the buffer does not contain any m_t messages, the return value is undefined. The second resume clause in the following fragment uses function qhead() to serve incoming *Request* messages in the order of their arrival.

```
receive (Request req) when (req.units <= units) {...}              /* first-fit */
receive (Request req) when ((qhead(Request)).units <= units) {...} /* FCFS */
```

Qempty(): Note that the guard in the preceding resume clause will be evaluated only if the message buffer contains a *Request* message. PARSEC also provides a function called **qempty**(m_t). The function returns *true* if the buffer does not contain any m_t messages, and returns *false* otherwise. For instance, the following receive statement gives higher priority to *Release* messages--it receives *Request* messages only when no *Release* messages are available.

```
receive (Request req) when (qempty(Release) && req.units <= units) {
        ...
}
or receive (Release rel) {
        units += rel.units;
}
```

Qlength(): PARSEC also provides a function called **qlength**(m_t). The function returns the number of m_t messages currently in the input queue (with timestamp <= to the entity's current simulation time). It might be used to balance service between two types of messages, as shown in the following example.

```
receive (Request req) when (qlength(Release) <= qlength(Request)) {
        ...
}
or receive (Release rel) when (qlength(Request) <= qlength(Release)) {
        ...
}
```

2.3.8 Compound Resume Conditions

In its most general form, a resume statement may include resume conditions associated with multiple message types as follows:

```
receive (Ma mvara, Mb mvarb, ..., Mn mvarn)
        statement;
```

Compound resume conditions have not been implemented, and probably won't be unless someone *really* wants them ($$$).

2.4 The Driver Entity

Every PARSEC program must include an entity called **driver**. This entity serves a purpose similar to the main function of a C program. Execution of a PARSEC program is initiated by executing the first statement in the body of entity **driver**. The **driver** entity takes the same **argc** and **argv** parameters as the C main function (except that argv must be declared *char** argv* because of PARSEC's requirement that array parameters have a constant size). Parameters recognized by the PARSEC runtime system will be removed from argc and argv before the driver is invoked.

2.5 Program Termination

A PARSEC simulation terminates in one of the following ways:

- *The simulation clock exceeds the maximum simulation time specified by* ***setmaxclock()****.*
- *All entities are suspended and no messages (including timeouts) are in transit.*
- *An entity executes an* ***exit()*** *or* ***pc_exit()*** *statement.*

When a termination condition is detected, each entity's (optional) finalize statement will be called. The entity may take appropriate actions before termination, including printing accumulated statistical data.

2.6 Clock Operations

In order to allow simulations to be executed over longer durations with fine grained clock values, the PARSEC system clock is implemented as a large integral type called **clocktype**. All clock operations make use of **clocktype** variables. The following functions are provided to manipulate the simulation clock:

- ***simclock(*void*)****: This function returns the value of the current simulation clock as a clocktype value.*
- ***setmaxclock(*clocktype*)****: This function sets the maximum simulation time to the value specified in the* clocktype *parameter. The simulation is terminated when the simulation clock exceeds this value.*
- ***atoc(*char*, clocktype**)**:* *Places the clocktype value represented by the string in the clocktype parameter.*
- ***ctoa(*clocktype, char*):** *Like sprintf, it prints the value of the clocktype parameter into the string parameter.*

Example:

```
clocktype time = (clocktype) 100000000;
setmaxclock(time);    /* set maximum simulation time to 100,000,000 */
```

The value CLOCKTYPE_MAX is predefined to contain the maximum value of **clocktype**. The default C type for **clocktype** is unsigned long, a 32 bit integer. The programmer may specify a larger type by using the compiler's –clock flag.

The example in Figure 2 illustrates the typical use of the *simclock()* function. The example shows the PARSEC code for a simple preemptible priority server which serves two types of jobs: *HighJob* - the high priority job, and *LowJob* - the low priority job. The server needs to sample the value of current clock whenever the service of a low priority job starts (line 25), is preempted (line 15), or re-starts (line 19). This is so the remaining service time of the low priority can be correctly computed whenever it is preempted by a high priority job.

2.7 Entity Scheduling

In a PARSEC program, an arbitrary number of entities may be mapped to a single processor. The execution of these entities is interleaved by the PARSEC scheduler. Entities are scheduled for execution based on the timestamps of their enabling messages.

An entity can be in one of four states: idle, ready, active, or terminated. An entity that has been terminated does not participate any further in the program. An entity that has not been terminated is said to be idle if its message buffer does not contain any enabling message. An entity whose buffer contains an enabling message is said to be ready; at any given point, multiple entities on a processor may be in the ready state. The scheduler selects the ready entity with the earliest enabling message for execution which then becomes active. An active entity relinquishes control to the scheduler only if it is terminated, or if it executes a hold or receive statement. In the latter case, if its buffer contains an enabling message it transits to the ready state (and is hence eligible to become active immediately); if not, it transits to the idle state. It is important to note that an active entity is self-scheduled: the scheduler cannot force it to relinquish control. In particular, an active entity that never executes a receive (or hold) statement, will never relinquish control to the scheduler.

```
message Highjob {
     int no_served;
};
message LowJob {
     int no_served;
};
entity Server()  {
    message LowJob lowjob;
    clocktype start_time, remaining_time;
    int       busy = 0;
    remaining_time = CLOCKTYPE_MAX;
    for (;;)  {
        receive (HighJob highjob)  {
            if (busy)
                remaining_time -= (simclock() - start_time);
            hold ((clocktype) expon());
            send HighJob{highjob.no_served+1} to next;
            if (busy)
                start_time = simclock();
        }
        or receive (LowJob 1j) when (!busy)  {
            lowjob = 1j;
            lowjob.no__served++;
            busy          = 1;
            start_time    = simclock();
            remaining_time= (clocktype) expon();
        }
        or timeout in (remaining_time)  {
            send lowjob to next;
            busy           = 0;
            remaining_time = CLOCKTYPE_MAX;
        }
    }
}
```

Figure 2: Priority Server

In the following example, the driver entity creates another entity called *first*, sends 50,000 messages to it, and then terminates. Since the entity does not execute a receive (or hold) statement, all 50,000 messages will be generated and delivered to entity *first*, before any of them can be processed by the destination entity.

```
entity driver(int argc, char** argv) {
      int i;
      ename first;
      first = new Sieve();
      for (i = 0; i <  50000; ++i)
            send Number{i} to first;
}
```

Programs such as this one can cause problems if they overflow system buffers, so programmers should take care when writing programs to avoid “source” entities like this.

2.8 Examples

2.8.1 Sieve of Eratosthenes

Constructs: message, ename, entity, new, send (2 forms), receive. This example programs the Sieve of Eratosthenes to generate all prime numbers less than 1000. The first instance of the *Sieve* entity is created by the driver and subsequent instances are created recursively such that the first number received by a new instance is a prime number. Subsequently the entity discards all multiples of its prime number and sends others to the next sieve in the pipeline. What follows is a PARSEC program to do this.

```
#include <stdio.h>

message Number {
    int number;
};

entity Sieve(int id) {
    ename next_sieve;
    int   myprime;

    receive (Number n) {
        myprime = n.number;
    }
    printf("Sieve number %d is for prime number %d\n", id, myprime);

    next_sieve = new Sieve(id + 1) at (id + 1);
    for (;;)
        receive (Number n) {
            if (n.number % myprime)
                send n to next_sieve;
        }
}

entity driver() {
    int   i;
    ename first;

    first = new Sieve(2) at (1);
    for (i = 3 ; i <= 1000; ++i)
        send Number{i} to first;
}
```

2.8.2 Topology Creation

Constructs: hold, setmaxclock. We present a series of examples that demonstrate the types of communication topologies that are commonly used in PARSEC programs. The examples use a simple entity type called *Delay*. The entity simply receives until it receives a message of type *Ping*. On receipt of the message, it suspends itself for a randomly distributed interval (in simulation time) and then forwards the message to one of its communication partners. The function *exp* used in the code returns a truncated value sampled from a random exponential distribution with the given mean value.

The first example demonstrates communication between a pair of *Delay* entities.

```
#include <stdio.h>

message Init {
    ename id;
};
message Ping {
    int originator;
    int trips;
};

entity Delay(int myno, int mean_time) {
    ename        next;
    message Ping p1;

    receive (Init i) {
        next = i.id;
    }

    while (1)
        receive (Ping p) {
            p1 = p;
            if (p1.originator == myno) {
                printf("\n Message No", myno,
                        "Number of round trips completed", p1.trips);
                p1.trips++;
            }
            hold(exp(mean_time));
            send p1 to next;
        }
}

entity driver() {
    ename e1, e2;
    e1 = new Delay(1,10);
    e2 = new Delay(2,10);
    send Init{e2} to e1;
    send Init{e1} to e2;
    send Ping{1,0} to e1;
    send Ping{2,0} to e2;
}
```

The next example sets up a ring containing 5 delay entities, where each entity knows the identity of its successor entity. The code for the entity is not changed; rather only the driver is changed to modify the communication topology. (Exercise: modify the program such that each entity knows the identity of both its predecessor and successor entities.)

```
entity driver() {
    ename prev;
    ename next;
    ename first;

    first = new Delay(0,10);
    prev  = first;

    for (i = 1; i < 5; i++) {
        next = new Delay(i,10);
        send Init{next} to prev;
        prev = next;
    }
    send Init{first} to prev;
    send Ping{0,0}   to first;
}
```

We now generalize the definition of a delay entity to allow multiple communication partners. The following entity called *Delay-Fork* is connected to *N* entities. On receiving a Ping message, the entity forwards this message to any one of the *N* neighbors with equal probability. The identities of the communication partners of the *Delay-Fork* entity are sent to it using an *Init-set* message. This message has two parameters: *count*, which refers to the number of communication partners for the entity; and array *id-set* that contains their identifiers. The declaration of the array *id-set* restricts the entity to a maximum of 10 communication partners.

```
message Init-Set {
    int   count;
    ename id-set[10];
};
message Ping {
    int hops;
};

entity Delay-Fork(int myno, int mean_time) {
    ename next[10];
    int   i, N;

    receive (Init-Set init) {
        N = init.count;
        for (i = 0; i < N; i++)
            next[i] = init.id-set[i];
    }

    while (1)
        receive (Ping p) {
            printf("\n Message No %d Number of hops completed %d",
                   myno, p.hops);
            hold(exp(mean_time));
            send Ping{p.hops+1} to next[urand(1,N)];
        }
}
```

The driver entity to set up a fully connected network of 5 delay-fork entities is as follows:

```
entity driver() {
    ename eids[5];
    int   i;

    for (i = 0; I < 5; i++)
        eids[i] = new Delay-Fork(i,10);

    for (i = 0; I < 5; i++)
        send Init-Set{5, eids} to eids[i];

    setmaxclock(10000);
}
```

3 The PARSEC Programming Environment

This section describes the fundamentals of writing, compiling, and executing PARSEC programs.

3.1 Using the Compiler

The PARSEC compiler, called *pcc*, accepts all the options supported by the C compiler, and also supports separate compilation. C programs (files with .c suffix) and object files (files with .o suffix) can also be compiled and linked with PARSEC programs. PARSEC programs are usually given a .pc extension.

3.1.1 Options

PARSEC compiler also supports the following options:

```
-protocol   Specify one of the synchronization algorithms:
              mpc             Message-passing C: ignores message timestamps
              cons            Conservative
              opt             Optimistic (not implemented yet)
-c          Generate ".o" and ".pi" files.
-E          Generate ".c" and ".pi" files.
-P          Inhibit line number translation. (Normally, the PARSEC
            compiler inserts line numbers into the intermediate C file
            so that compiler and runtime errors report
            the correct line in the PARSEC file.)
-env        Show environment names set for pcc.
              PCC_DIRECTORY - installation directory for PARSEC.
              PCC_CC - C compiler for PARSEC to use.
              PCC_LINKER – linker for PARSEC to use.
              PCC_CC_OPTIONS - options to pass to the C compiler.
              PCC_PP_OPTIONS - default options for the PARSEC compiler.
              PCC_LINKER_OPTIONS – default options for C linker.
-pcc_directory      – use specified PCC_DIRECTORY.
-pcc_cc             - use specified C compiler.
-pcc_linker         - use specified linker.
-pcc_pp_options     – use these options for preprocessing.
-pcc_cc_options     – use these options for compiling.
-pcc_linker_options – use these options for linking.
-ini        Save the auto-generated initialization file.
-ff         Enable compilation of friend functions.
–stack      Change the default stack size for entities.
-help       Show compiler options.
-v          Verbose compilation.
-V          Show the version number of the compiler.
-user_main  Rename the main function to parsec_main, and link with user’s main
            function.
-shared_lib Rename the main function to parsec_main and create a shared library.
-clock      Set the default representation for clocktype.  Valid options are:
              unsigned          unsigned long, a 32 bit integer type (default)
              longlong          long long, a 64 bit integer type
```

The following examples illustrate how to compile PARSEC programs on a sequential architecture.

% pcc -o example example.pc

This generates an executable file *example* in the current working directory.

% pcc example.pc

This generates an executable file *a.out* in the current working directory.

% pcc -o example example.pc xxx.c yyy.o

This generates an executable file called *example* in the current working directory. The file *example* is compiled and linked with xxx.c and yyy.o.

3.1.2 Separate Compilation

PARSEC supports separate compilation of entities. Entities defined in one file and used in a second file must be declared *extern* in the second file.

The following example illustrates the use of entity *Sort* as an extern entity:

Example:

```
file1:
       entity Sort (int n, int a[20] )
              ...

file2:
       extern entity Sort(int, int[20]);  /* This declaration cannot be omitted */

       entity driver() {
              ename s1;
              int x[20];
              s1 = new Sort(20, x);
       }
```

Compiling separate files is as easy as this:

% pcc client.pc server.pc

This compiles two PARSEC files - client.pc and server.pc - and generates *a.out* as the executable. (One of these files must contain the driver entity.)

Remarks:

1. Besides a .o file, the PARSEC compiler creates a .pi file for each source file. The .pi file contains information about the message types used in the file and must be visible to the compiler at the link stage, unless all the message types in it are also used in other files.

3.2 Parallel Compilation and Execution

In order to produce an executable that can be run on a parallel architecture, the PARSEC program needs to be compiled with the '-sync protocol' flag. Currently supported protocols are *cons* -parallel conservative, *opt* -parallel optimistic, *mpc* -parallel without synchronization. A parallel optimistic protocol is in development. On shared memory architectures, the program will be made to use POSIX threads. On distributed memory machines, such as the IBM SP2, MPI will be used for communication. To use MPI, it first needs to be installed on the available parallel architecture (There are several public domain implementations of MPI. One such implementation, MPICH, is available from *http://www.erc.msstate.edu/mpi/*). MPI commands are used to execute the compiled PARSEC program on the parallel architecture (see the documentation on the particular MPI implementation you are using).

Several different conservative algorithms are available, including a null message protocol (the default), a conditional event protocol, a combined conditional event and null message protocol, and the Ideal Simulation Protocol (ISP). ISP calculates a lower bound on the execution time of a parallel simulation by first collecting a valid trace of a program run, then executing that program a second time with advance knowledge of the order of message execution. It can then run a simulation without any overhead. The conservative runtime library recognizes the following command line options for selecting a protocol, all of which must follow a -- (double dash):

```
-np n        create n threads
-null        use the null message algorithm (this is the default)
–cond        use the conditional event algorithm
-sync        use a synchronous protocol
–anm         use of combination of the preceding two algorithms
–trace       collect a runtime trace of the simulation
-isp         use the ISP protocol and the tracefiles generated with -isp_trace
-barrier     enable barrier synchronization for –cond, -sync, and -anm
-dest        activate destination specific lookahead
```

% pcc -o example -sync cons example.pc
% example -- -np 8 -cond

This generates an executable file *example* in the current working directory, and runs it with 8 threads (distributed to 8 nodes of an SMP) using the conditional event protocol.

3.3 Debugging PARSEC Programs

Debugging support for PARSEC programs is provided through traditional source-level debuggers like *dbx* or *gdb* available with UNIX to step through the PARSEC program. Some additional debugging features available in Maisie haven't been implemented yet, but are described briefly here.

3.3.1 Diagnosing Compiler Errors

A PARSEC program is translated into a C program, which is then compiled using a C compiler. During the translation process, the PARSEC compiler marks line numbers in the C file for use in the C compiler and debugger. Sometimes those line numbers don't match perfectly because the PARSEC compiler considers braces ({}) and semicolons to be terminals. A statement such as:

```
if (a)
       a = false;
```

will be recorded as a single line. Some error messages will therefore be off by one line.

3.3.2 Source Level Debugging

A PARSEC program is translated into a C program, which is then compiled using a C compiler. It is possible to debug at the level of PARSEC source using a standard Unix C debugger. The program needs to be compiled using the '-g' flag. The entity parameters and local variables are renamed according to the following conventions:

- Entity **X** $\Rightarrow$ *function ``* ***MC_entity_X***''*. For example, in dbx, the command ``stop in* ***MC_entity_X***'' *enables a break point whenever an instance of entity* ***X*** *is scheduled. (The debugger will not distinguish between different instances of each entity type, but you can use its local variable or parameter values to distinguish between instances.)*

- Entity parameters and local variables **X** $\Rightarrow$ `` ***X***''. *For example, in dbx, the command ``print **X**'' prints the value of **X** to the terminal.*

Below, we show a sample PARSEC program, and how it is compiled, executed, and debugged in a dbx session. As seen in the program, line 3 will cause the program execution to have segmentation violation due to an incorrect array reference. Lines 20-23 show how parameter *a* of entity *XX* and local variable *b* can be accessed directly.

```
[1] entity XX (int a) {
[2]   int *b;
[3]   b[a]++;
[4] }
[5]
[6] entity driver() {
[7]   new XX(5);
[8] }
```

Figure 3. Debugging: the sample program

```
[9]  % pcc -g sample.pc
[10]  % a.out
[11]  Segmentation fault
[12]  % dbx a.out
[13]  Reading symbolic information...
[14]  Read 1482 symbols
[15]  (dbx) run
[16]  Running: a.out
[17]  signal SEGV (no mapping at the fault address) in MC_entity_XX ...
[18]    3   b[a]++;
[19]  (dbx) print b
[20]  b = 0
[21]  (dbx) print a
[22]  a = 5
```

Figure 4. Debugging using dbx

Remarks:

2. In order to use dbx, cc must be the default C compiler. See Section 3.1 for details on setting the default C compiler.

3.3.3 Common PARSEC Runtime Errors

Runtime errors/warnings are detected by PARSEC and appropriate messages are send to *stderr*. The most common errors are related to stack allocation. Please refer to the troubleshooting section (Appendix D) for further information. Some of the other errors/warnings are:

- *"Run out of memory for a message."* Memory has been exhausted, possibly due to a deadlock, an infinite loop, or an imbalance in message processing.
- *"Error: Trying to send messages to remote entity."* A sequential program is attempting to send a message to a remote processor, probably indicating that an ename variable is uninitialized or has become corrupted.
- *"Error: Trying to receive messages from remote entity."* Same as above, but less common.
- *"Thread Local Storage Key Create Error."* Error in creating Windows NT threads.

- *"Fail to create NT thread."* Ditto.
- *"*** PARSEC error. Failed in …"* Pthread setup error.
- *"*** PARSEC error. Failed to create threads."* Pthread setup error.
- *"Unrecognized option ... Ignored."* Unrecognized command line option following the '--'.
- *"Entity 'type' (node, id) did not fit the allocated stack space … Specify bigger stack size for this entity."* The local data of the entity is too large to fit in the space allocated for it. A larger size can be specified with the **stacksize** keyword. There are several variations of this error message.
- *"Too many different types of messages. Recompile the runtime to support more message types."* By default, the maximum number of message types the Parsec system can support is 64. The number can be increased, but it has a detrimental impact on performance.
- *"Run out of memory in setting entity parameters."* Apparently, the parameters for this entity are HUGE.
- *"Wrong NEW_ENTITY_ACK."* An error in creating a new entity has occurred. Please report this to the Parsec development team.
- *"Unrecognized Remote Message."* Somehow the message has become corrupted, possibly due to memory mismanagement in the program.
- *"(Node xxx): terminating the execution."* Several types of errors generate this message, clock overflow for example.

3.3.4 Conservative PARSEC Runtime Errors

The following errors may occur when using the conservative parallel synchronization algorithms. Many of these errors concern mistakes in lookahead specification which can lead to causality errors.

- *"Entity* 'type' *(*node, id*) tried to send a message to entity '* type*' (*node, id*) with timestamp (*time*), which is lower than the previous EOT value (* time*)."* This is a causality error, the most likely cause of which is a previous call to *setlookahead* with a too-large delta or ceiling.
- *"Entity 'type' (node, id) reduced its EOT value for entity 'type' (node, id) from time to time."* An entity has reduced its lookahead value without advancing its time by an equal amount. This error means that the original lookahead value may have been too high, leading to a possible causality error for the receiver.
- *"Entity 'type' (node, id) reduced its EOT value from time to time."* The entity has reduced its lookahead, which may lead to one of the previous two error conditions. There are several variations of this error message.
- *Entity 'type' (node, id) tried to send a message to entity 'type' (node, id), which is not in the destination set."* Each entity maintains a destination set of entities to which it sends messages. If this list is not accurate, the destination entity may execute messages out of order, or may become deadlocked.
- *"Entity 'type' (node, id) received a message from entity 'type' (node, id), which is not in the source set."* The converse of the previous error. Each entity must also have a correct and complete source set, or it may process messages (from entities not listed in the source) in an incorrect order.

- *"Sent a message to entity 'type' (node, id) with timestamp below the global ECOT."* Whereas the EOT messages shown earlier occur in the null message and accelerated null message protocols, this happens in the conditional event protocol. It also results from a lookahead value which has been set too high.
- *"Error in add_source(). The argument is incorrect."* Most likely, the ename value has not been initialized.
- *"Error in del_source(). The argument is incorrect."* Same as above.
- *"Error in add_dest(). The argument is incorrect."* Same as above.
- *"Error in del_dest(). The argument is incorrect."* Same as above.
- *"Run out of memory to update a destination hash table. Try the conditional event protocol or more nodes."* The processor ran out of memory trying to create space for an entity's destination set. The conditional event protocol does not require source and destination sets, so it may require less memory.
- *"Run out of memory to update a source hash table. Try the conditional event protocol or more nodes."* Same as above.
- *"Failed open a trace file 'filename'.* An error in running ISP.
- *"Failed reading 'filename'."* Unable to load an ISP trace file.
- *"No entity mapped on this node."* An ISP warning that your program doesn't use all the processors.
- *"Not enough memory for storing ISP data."* Dang!
- *"Error in ISP entity name file: filename."* A corrupt ISP trace file. Sometimes the exact record number is reported.
- *"Error in ISP message event file: filename."* Similar to the above, the error is in the event listing rather than the entity listing.
- *"Failed allocating memory for storing ISP data."* Dang again!
- *"Failed making a trace file 'filename'."* Couldn't create the trace file. Permission problem?
- *"Failed writing a trace file 'filename'."* Couldn't write to the trace file. Out of disk space? Out of quota? ISP requires a lot of disk space.

4 Parallel Simulation Requirements

Note: The parallel PARSEC runtime object files are not included in the default distribution. This is because parallel simulation is hard, very hard, and providing support for new users is very time-consuming, especially for me, since I answer most of the questions. To acquire parallel PARSEC usually requires that you first demonstrate proficiency in sequential PARSEC simulation, while following the guidelines given here to ease parallelization. Please contact Dr. Bagrodia (rajive@cs.ucla.edu**) to discuss upgrading to the full version.**

PARSEC is currently implemented using several different synchronization algorithms - sequential (global event list), parallel conservative (using null messages, conditional events, or a combination of these two), and ISP (for Ideal Simulation Protocol). Several optimistic simulation protocols are under development and should be ready soon. In principle, a PARSEC simulation program that is executed using a sequential simulation algorithm will also execute correctly using a parallel algorithm. However, in practice, certain guidelines must be followed while writing the sequential model to guarantee portability and efficient execution in the parallel environment. This section gives some brief recommendations for maintaining parallel compatibility and lists some functions provided by PARSEC for this purpose.

4.1 Restrictions

Parallel PARSEC programs must conform to the following restrictions. These restrictions apply to almost any message-based parallel language and are not specific to PARSEC. The PARSEC compiler might not generate errors or warnings for the following problems because they are legal sequential programs, but a program violating these rules will most likely abort or produce incorrect results when run in parallel.

- *Shared/Global Variables: A parallel PARSEC simulation should not use `global' variables. This includes function and file scope static variables. Read-only variables initialized at the beginning of the simulation can be global, but must be initialized separately on each processor (on a distributed memory architecture). We cannot emphasize this enough.* ***DO NOT USE GLOBAL VARIABLES!***
- *Pointers cannot be passed in messages or entity initialization parameters. Note that this implies that dynamic linked-list data-structures cannot be passed. Arrays must be used instead.*

4.2 Partitioning

For parallel implementations, the simulation model must be partitioned by allocating entities among the processors. PARSEC allows the programmer to specify the specific node on which an entity will be created by using the **at** option during entity creation. In general, the model should be partitioned such that message communication between nodes is minimized, with balanced computation load.

4.3 Requirements for Conservative Simulation

There are two primary changes required to convert a PARSEC simulation for use with the conservative protocols. First, the communication topology must be specified, and second, lookahead must be provided. These concepts are explained in the following subsections.

4.3.1 Communication Topology

The runtime system needs to be informed about the communication topology between the entities in the system. This information is used by the system to do the necessary synchronization. It is required for the null message protocol (-null), the conditional event protocol (-cond), and the accelerated null message protocol (-anm). The runtime system implicitly maintains a *source-set* and a *destination-set* for each entity. For each entity, its *source-set* is the set of all the entities that send messages to it, and the *destination-set* is the set of all the entities that it sends messages to. Every entity needs to inform the runtime of the entities it wants to be part of its source or destination set. This is done by using the following PARSEC system calls:

- add_source(*e*): Add ename *e* to my source set.
- del_source(*e*): Delete ename *e* from my source set.
- add_dest(*e*): Add ename *e* to my destination set.
- del_dest(*e*): Delete ename *e* from my destination set.

Exceptions:

- *If an entity e1 creates entity e2, then at the time of entity creation, the runtime system* automatically *places e2's ename in e1's destination set, and e1's ename in e2's source set. If this link is not necessary for the given simulation, user can delete this link, by using del_dest in the parent, and del_source in the child, immediately following the entity creation.*

- *An entity does not need to add itself to its source or destination set, even if it intends to send messages to itself.*

Topology setup can be tricky. If one entity sends a message to a second entity before the second entity has executed add_source(), an error will be reported because the second entity may have already advanced its local clock beyond the timestamp of the incoming message. The easiest way to complete the topology setup correctly is to use a third party entity (usually the driver) and a simple handshaking procedure, as shown in the following example.

```
message InitDest { ename source; };
message InitSource { ename dest; };
message SetupComplete {};
message Start {};

entity Dest(ename creator) {

    receive (InitDest init) {
        add_source(init.source);
    }

    add_dest(creator);
    send SetupComplete{} to creator;

    receive (Start s);
    del_dest(creator);
    del_source(creator);

    /* topology setup complete, start normal operation */
}

entity Source(ename creator) {

    receive (InitSource init) {
        add_dest(init.dest);
    }

    add_dest(creator);
    send SetupComplete{} to creator;

    receive (Start s);
    del_dest(creator);
    del_source(creator);

    /* topology setup complete, start normal operation */
}

entity driver(int argc, char** argv) {

    ename source, dest;

    source = new Source(self);
    dest   = new Dest(self);

    add_source(source);
    add_source(dest);

    send InitSource{dest} to source;
    send InitDest{source} to dest;

    receive (SetupComplete sc); /* once from source */
    receive (SetupComplete sc); /* and once from dest */

    send Start{} to source;
    send Start{} to dest;

    /* topology setup complete, terminate */
}
```

4.3.2 Lookahead

Lookahead is the amount of time between *now* (the current time of an entity) and the time when the entity will send its next message. For example, if an entity A is at time 10, and it won't send a message until time 20, then its lookahead is 10. It also means that the entities in A's destination set can safely process any message with a timestamp less than 20 because it won't get a new message from A until then. If lookahead is large, the performance of the simulation will be better, because more events can be processed in parallel. Lookahead must be specified in conservative PARSEC programs, and is normally specified as the minimum time between the receipt of the next message, and the first message sent as a result. Consider the following PARSEC code:

```
...
receive (Job job) {
       hold(5);
       send job to next;
}
...
```

In this code segment, the lookahead is 5, because the entity will not send a new message until 5 time units after it receives a Job message.

Lookahead is set by using the two following PARSEC system calls:

- *setlookahead(*delta, ceiling*): sets the current lookahead value to* delta *and sets* ceiling *as a maximum value for lookahead.*
- *setdestlookahead(delta, ceiling,* dest*): sets the lookahead for a particular member of the destination set.*

The setdestlookahead call is used to set different values of lookahead with respect to different destination entities, and must be used in conjunction with the - dest runtime parameter. *The PARSEC runtime resets the values of lookahead* (to 0, CLOCKTYPE_MAX) *after every **hold** statement.* setlookahead and/or setdestlookahead must be set before the first receive statement, and must be reset after each hold statement, and whenever the lookahead changes.

To run the preceding code segment with a conservative algorithm, we must add the setlookahead call as shown here:

```
...
setlookahead(5, CLOCKTYPE_MAX);
receive (Job job) {
       hold(5);
       send job to next;
}
...
```

The second parameter of the setlookahead function is a ceiling value. The ceiling value is used when a receive statement contains a timeout clause, and is typically set to the current simulation time plus the value of the timeout interval, as in the following example:

```
message LowJob lowjob;

setlookahead(5, simclock() + 10);
receive (HighJob highjob) {
       hold(5);
       send highjob to next;
```

```
    }
    or timeout in (10) {
           send lowjob to next;
    }
```

The lookahead value is used to compute the earliest time at which this entity might send a message. In the preceding example, if a HighJob message arrives, the entity will not send a message until 5 time units later, so the lookahead is 5. Suppose that this receive statement is issued at time 0 and that 6 time units pass without a HighJob message arriving. If a HighJob message arrives, a message will be sent at time 11, but if no HighJob message arrives, the receive statement will timeout and a message will be sent at time 10. So the earliest time that this entity will send a message is time 10. The ceiling value is used to express the timeout condition. The ceiling should be set to CLOCKTYPE_MAX whenever there is no timeout clause in a receive statement (or if no message is sent immediately after the timeout).

For the most efficient, trouble-free execution, there should be non-zero lookahead for every message *and* after the timeout, as shown in the following example. Even the message after the timeout is sent only after a delay of 1, and the ceiling is adjusted accordingly.

```
    message LowJob lowjob;

    setlookahead(5, simclock() + 11);
    receive (HighJob highjob) {
           hold(5);
           send highjob to next;
    }
    or timeout in (10) {
           send lowjob to next after 1;
    }
```

A more complete example of a conservative program is shown in Figure 5. It is a refinement of the priority server example given in Figure 2. Two things have been added, a small code segment for setting up the topology, and some code for setting the lookahead. These two sections will be discussed in the following paragraphs.

There is an additional message type called *Initialization*, which is used to set up the communication topology. The PriorityServer entity is assumed to be in a network of servers, not necessarily all of them priority servers. It can send/receive jobs to/from any of the servers. Thus, when it receives the initialization messages, it adds each server to its source set and its destination set. The driver entity is automatically added to the source set of each entity when the driver creates it, but the PriorityServer does not send messages to the driver, so need not add it to its destination set.

In order to specify lookahead, we must employ a common trick. We precompute the service time of the next job by running the random number generator before entering the receive statement, rather than after receiving a message. (Compare the two figures.) This allows us to specify the service time of the next job as the lookahead. The ceiling we place on the lookahead is the remaining service time of the previous low priority job (if there is one), or CLOCKTYPE_MAX otherwise.

4.4 Requirements for Optimistic Simulation

The optimistic synchronization algorithm is not yet available for PARSEC.

```
message Initialization { ename servers[10]; };
message Highjob        { int no_served; };
message LowJob         { int no_served; };

entity PriorityServer(int number_of_servers) {
    clocktype      remaining_time, service_time, start_time;
    ename          servers[10];
    int            busy, server;
    message LowJob lowjob;

    receive (Initialization init) {
        for (server = 0; server < number_of_servers; server++) {
            servers[server] = init.servers[server];
            add_source(servers[server]);
            add_dest(servers[server]);
        }
    }

    busy           = false;
    remaining_time = CLOCKTYPE_MAX;
    service_time   = expon(); /* precompute the service time */

    for (;;) {

        setlookahead(service_time, remaining_time);

        receive (HighJob highjob) {
            if (busy)
                remaining_time -= simclock() - start_time;
            hold(service_time);
            service_time = expon(); /* precompute the next service time */
            send HighJob{highjob.no_served + 1} to servers[/*random choice*/];
            if (busy)
                start_time = simclock();
        }
        or receive (LowJob lj) when (!busy) {
            lowjob = lj;
            lowjob.no_served++;
            busy           = true;
            start_time     = simclock();
            remaining_time = service_time;
            service_time   = expon(); /* precompute the next service time */
        }
        or timeout in (remaining_time) {
            send lowjob to servers[/*random choice*/];
            busy           = false;
            remaining_time = CLOCKTYPE_MAX;
        }
    }
}
```

Figure 5. Conservative Priority Server

Appendix A: PARSEC Library Reference

This section contains brief descriptions of the library functions provided for use in PARSEC programs.

A.1 Clock Operations

- ***simclock(void)***: *This function returns the value of the current simulation clock as a* clocktype *value.*
- ***setmaxclock(*clocktype*)***: *This function sets the maximum simulation time to the value specified in the* clocktype *parameter. The simulation is terminated when the simulation clock exceeds this value. item* ***hold(*clocktype*)***: *advances the current simulation clock by the specified amount.*
- ***atoc(*char*, clocktype****)***: *Places the* clocktype *value represented by the string in the* clocktype *parameter.*
- ***ctoa(*clocktype, char*)**: *Like sprintf, it prints the value of the* clocktype *parameter into the string parameter.*

A.2 Conservative Simulation Functions

- ***add_dest(ename)***: *adds the specified entity to the destination set of the current entity*
- ***add_source(ename)***: *adds the specified entity to the source set of the current entity.*
- ***del_dest(ename)***: *removes the specified entity from the destination set of the current entity.*
- ***del_source(ename)***: *removes the specified entity from the source set of the current entity.*
- ***setlookahead(clocktype, clocktype)***: *sets the lookahead for the current entity.*
- ***setdestlookahead(clocktype, clocktype, ename)***: *sets the lookahead for a specific destination.*

A.3 Random Number Generators

PARSEC is based on ANSI C, and ANSI C doesn't include any (good) random number generation functions, so these functions are provided. They use the same algorithm as the 48 bit equivalents (erand48, etc.) on Sun Solaris, but are somewhat more efficient because they don't provide all the options provided by Sun.

- ***double pc_erand(unsigned short[3])***: *returns a value in the range [0.0, 1.0)*
- ***long pc_jrand(unsigned short[3])***: *returns a value in the range* $[-2^{31}, 2^{31})$
- ***long pc_nrand(unsigned short[3])***: *returns a value in the range* $[0, 2^{31})$

A.4 Miscellaneous

- ***CLOCKTYPE_MAX***: *a constant* clocktype *which contains the maximum value of the* clocktype *type (whether it's represented as a float, a long, a long long, or whatever).*
- ***ENULL***: *a constant ename variable used to designate a null value. Enames are not initialized by default. They must be explicitly set to ENULL.*
- ***int ename_cmp(ename, ename)***: *compares two* ename *variables and returns a non-zero value if they are equal.*

- ***int ename_valid(ename)****: returns a non-zero value if the ename is not ENULL.*
- ***int pc_num_nodes()****: returns the number of parallel processors being used for this program run.*
- ***setmaxclock(clocktype)****: sets a limit on the length of execution.*
- ***pc_exit(int)****: the normal exit() function causes problems in an MPI program. This is a safer alternative.*
- ***pc_printf(char*,…)****: a parallel version of printf, it prepends the processor number to each string.*
- ***pc_fprintf(char*,…)****: a parallel version of fprintf.*
- ***entity_yield()****: forces a context switch to a different entity with an earlier message, if one exists.*
- ***pc_print_runtime()****: prints the execution time of the program (so far).*

Appendix B: New Things in PARSEC

The PARSEC system is a major upgrade from the last version of Maisie. The syntax has been changed to create a more readable, intuitive, language. The redesign also makes the system more flexible, extensible, and less error prone. This section briefly lists the differences between Maisie and PARSEC, and the next section will give details on converting Maisie programs into PARSEC.

B.1 Performance

The PARSEC compiler is smaller and much faster than the Maisie compiler. Sequential PARSEC simulations have been up to an order of magnitude faster than identical Maisie programs.

B.2 Availability

PARSEC has been implemented using a portable kernel, works on Linux as well as most commercial flavors of Unix, and has also been tested on Windows NT.

The parallel implementation of Maisie was not included in the normal distribution. Parallel PARSEC will be part of the standard distribution.

B.3 Implementation Changes

Two major implementation changes have been made, which could have a significant effect on program organization:

1. Entities are each given a private stack space. *This allows users to put* ***receive*** *statements in any ordinary function or friend function*, which makes the code much more modular, and easier to evolve from simulation to application.
2. PARSEC uses an efficient one-pass compiler. This forces the syntax to be somewhat more restrictive.

B.4 Syntax Changes

The most obvious differences are in the syntax. Instructions for converting Maisie programs into PARSEC are available in the next section.

- *PARSEC uses ANSI style declarations for entities.*
- *Maisie's* ***wait ... until*** *syntax has been replaced with* ***receive****.*
- *invoke ... with has been replaced by send ... to*
- *Message variables in a PARSEC* ***receive*** *statement are read-only.*
- *Message types must be declared globally in PARSEC, not just in the receiving entity.*
- *Friend Function declarations must appear before the entity definition.*
- *PARSEC adds a* ***finalize*** *block at the end of the entity body.*
- *PARSEC fully supports nested* ***receive*** *statements.*

Appendix C: Modifications Required for Compatibility

This section gives examples of the differences between Maisie and PARSEC and the changes required for conversion. The primary changes include the use of the ANSI C style, the replacement of the invoke and wait syntax with send and receive commands, and further restrictions on the order and placement of declarations.

C.1 Send and Receive

The syntax for sending and receiving messages has been revamped. We believe the new syntax is more intuitive and more readable. Maisie's **invoke ...with** construct has been replaced by PARSEC's **send ...to** syntax, as shown here:

```
ename e;
message M{} m;
```

Maisie:

```
invoke e with M = m;
invoke e with M{};
```

PARSEC:

```
send m to e;
send M{} to e;
```

Similarly, Maisie's **wait until** syntax has been changed to PARSEC's simpler **receive** syntax. The keyword **when** replaces the keyword **st** to indicate a guard.

```
ename e;
message M{int I;};
```

Maisie:

```
wait 5 until {
        mtype(M) st (msg.M.i < 10) {
                . . .
        }
        or mtype(timeout) {
                . . .
        }
}
```

PARSEC:

```
receive (M m) when (m.i < 10) {
        . . .
}
or timeout in (5) {
        . . .
}
```

(Note that a compound Maisie **wait** statement required an enclosing block, but the PARSEC **receive** statement is structured like an if-else-if chain.)

C.2 Receive Message Variables are Constant

Message variables declared in a receive statement are constant, read-only variables. They can not be modified by the program. In order to modify the variable, it must first be copied into another variable. (This was done for efficiency in execution.)

Maisie:

```
wait until mtype (Ping) {
        msg.Ping.trips++;
        invoke next with msg.Ping;
}
```

PARSEC:

```
message Ping ping;

receive (Ping p) {
        ping = p;
        ping.trips++;
        send ping to next;
}
```

C.3 ANSI C Style Entity Declarations

Entities are now declared in the ANSI C style, with full parameter specification for both extern entity declarations and entity definitions and parentheses instead of braces.

Maisie:

```
extern entity Manager{};
entity Sort {n, a}
        int n;
        int a[20];
{ . . .
```

PARSEC:

```
extern entity Manager(int);
entity Sort (int n, int a[20]) {
        ...
```

C.4 Random Number Generators

The switch to ANSI C has the consequence that the popular unix random number numbers, such as erand48 and drand48, which are not included in the ANSI standard, must be explicitly declared in order to be used. It is insufficient to include stdlib.h. We recommend using the random number generators provided with PARSEC. See Section A.3.

Maisie:

```
#include <stdlib.h>
```

PARSEC:

```
extern double drand48();
```

C.5 Message Declarations

In Maisie, only entities which received a message of say, type *A* needed to declare message type *A*. Entities which sent, but did not receive, messages of type *A* did not need to declare it. This situation led to a very common programming error wherein the sending and receiving entities either separately (and differently) defined type *A*, or the sender (which didn't need to declare the message) sent the wrong number of arguments or put them in the wrong order. In PARSEC, message types declarations must be visible to both the sender and the receiver, and must be globally unique throughout the program. The compiler verifies the number and type of arguments.

```
message M{};
entity A() {
      send M{} to b;
}
entity B() {
      receive (M m) ...
}
```

C.6 Friend Functions

Friend functions must now be declared before the entity body.

Maisie:

```
entity Sort{} {
      int sorting();

      . . .
}

int Sort::sorting() {
      . . .
}
```

PARSEC:

```
entity Sort();
int Sort::sorting();

entity Sort() {
      . . .
}
```

C.7 endsim and finalize

Maisie included a special message type called **endsim,** which was delivered automatically to each entity when the maximum simulation time was reached. However, there was difficulty in writing correct programs because every wait statement needed to also wait for endsim, and every hold statement had to be replaced with an equivalent wait for endsim. PARSEC replaces endsim with a **finalize** block which, if present, is called for each entity when that entity exits. A representative example of the necessary change is as follows:

Maisie:

```
wait until {
      mtype (Request) {
            . . .
      }
      or mtype (endsim) {
            /* print results */
      }
}
```

PARSEC:

```
receive (Request req) {
       . . .
}

finalize {
       /* print results */
}
```

C.8 Library Function Changes

Maisie	PARSEC
maxclock(char *)	setmaxclock(clocktype)
nnodes()	pc_num_nodes()
print(char*, …)	pc_printf(char*, …)

C.9 Maisie features which have not been implemented (yet)

- *The trace facility and* ***trace_msg*** *.*
- *The print_msg command.*
- *Compound resume conditions.*
- *The histo and basic_stats library entities.*

C.10 Incompatibilities between PARSEC versions 1.0 and 1.1

- ***terminate()*** *replaced by* ***pc_exit()***
- ***nnodes()*** *and* ***num_nodes()*** *replaced by* ***pc_num_nodes()***
- ***-sync seq*** *removed. It wasn't necessary anyway, since it was the default.*
- ***-arch*** *removed. The choice is now automatic, based on the type of machine.*
- ***ctoi() and ctof()*** *and all other clock functions (clockadd, etc.) removed.*

Please also read the readme.txt file that comes with the distribution for more information.

Appendix D: Troubleshooting

D.1 Stacksize Problems

Each entity is now given a private stackspace, which is used to store the entity's local variables as well as for making function calls. Creating a private stack space for each entity allows receive statements to appear in functions. The default stacksize for an entity is 200KB, but can be modified during compilation with the -d flag.

There are two common problems with entity stack sizes, it is either 1) too large, or 2) too small. When the stacksize is too large, the machine will run out of memory. When the stacksize is too small, the program will report an error and abort.

The "too small" case occurs when an entity has a large amount of local data, such as a large multidimensional array, or when it makes a series of deeply nested recursive function calls. In this case, the programmer can specify a larger stack size for the problem entity, using the **stacksize** option on entity definition.

Often in the "too large" case, the problem is not with the program, but with the unix shell. On Solaris machines, for example, the system defaults to an 8 MB limit on stack space, which means only 40 entities can be created. On Solaris and similar systems, the user can use the " unlimit" shell command to remove this limit. On linux, apparently only the root user can unlimit stacksize without recompiling the kernel.

However, in large simulations with hundreds of fine-grained entities, it may be necessary to reduce the minimum stack size for an entity. To do this requires modifying one or more preprocessor directives and recompiling the runtime system. Since the source is not provided in the default distribution, this requires a special request to parsec@cs.ucla.edu. The current minimum stacksize is 20KB, and we have found it very dangerous and unpredictable to go below this limit. The default stack size can also be adjusted with the -stack compiler flag, but not below the 20KB limit.

D.2 Bug Report and System Support

We are able to answer questions about this software as our time allows. Please submit bug reports using the bug reporting form on our web page - http://pcl.cs.ucla.edu/projects/parsec/, and feel free to submit suggestions on the PARSEC bulletin board at the same web address.

Appendix E: Getting PARSEC

A registration form for downloading the latest version of PARSEC is available on the Parallel Computing Laboratory web site at http://pcl.cs.ucla.edu/projects/parsec/. Installation instructions are included with the distribution.