Performance Efficient Parallel Programming in MPC

D. Vrsalovic, Z. Segall, D. Siewiorek, F. Gregoretti, E. Caplan, C. Fineman, S. Kravitz, T. Lehr, M. Russinovich)

13 July 1988

CMU-CS-88-167

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Multiprocessor C (MPC) a C language preprocessor, which assists a programmer in building efficient parallel programs, is described. MPC provides the programmer with a virtual implementation machine. We also present the *Consistent Abstract Shared Data Type Implementation Machine (CASDTIM).* PIE embraces the concept of "programming for observability" in which the user makes use of visual tools to aid in the development, testing and debugging of his application. Extensive examples written in MPC are presented in the Appendices.

This research has been supported in part by the Ballistic Missile Defense Advanced Technological Center under contract DASG-60-86-C-0015. The views and conclusions contained in this paper are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of BMDATC, Carnegie Mellon University or the U.S. Government.

Table of Contents

List of Figures

1. Introduction

Modem parallel systems are designed to achieve two main goals: high performance and increased availability. Both goals can be achieved via parallel use of the system resources, but one should be aware of the fact that the use of parallelism brings increased system complexity. Conventional system design tools tend to cope with increased complexity of the designs by imposing a layered hierarchy utilizing the concept of abstraction. Intensive use of communication and synchronization is required to implement these abstractions. While abstractions simplify design time complexity, they are a major source of run-time performance degradations.Performance degradation will usually arise in one of four forms: 1. delays due to contention on common resources, 2. delays due to synchronization overhead, 3. increased load due to unfavorable parallel decomposition, and 4. unbalanced load on the resources in the system. While the first two forms of degradation have been widely investigated, we know of no models today which are capable of analyzing the latter two. These forms of degradation remain a challenge for future research.

The role of models is to predict performance bottlenecks during the design process, and thus reduce time spent during the development. Due to the simplifying nature of models, we cannot expect them to predict all of the bottlenecks. Thus special tools are required to assist the developer in detecting further sources of performance degradation. Unfortunately, these tools require run-time data collection which, in practice, is invasive. Invasive tools not only add to the workload but also can artificially introduce new bottlenecks.

Once programmer has produced his best design, the role of minimizing remaining performance degradation falls upon run-time support system. The PIE project, as depicted in Figure 1-1 supports the entire design process from modeling(i.g. prevention), to monitoring(e.g. detection), to run-time(e.g. avoidance).

PIE [Segall 85] views parallel processing in the context of "implementation machine" (IM) models. IMs are the user templates which supply low level process synchronization and communication details for the programmer. The user can thus concentrate on algorithm design and implementation to a greater degree than previously possible.

The PIE system's approach tends to eliminate performance degradations due to classical structured approaches by introducing "virtual" rather than "physical" layers. The structure is available during program development time when the abstractions are required to assist in understanding the complexity. By run-time, however, the structure has been flattened and removed yielding higher performance parallel programs.

PIE also embraces the concept of "programming for observability" in which the user makes use of visual tools to aid in the development, testing and debugging of his application [Gregoretti 85] [Snodgrass 82]. During development, the PIE system incrementally builds a view of the user program's semantic

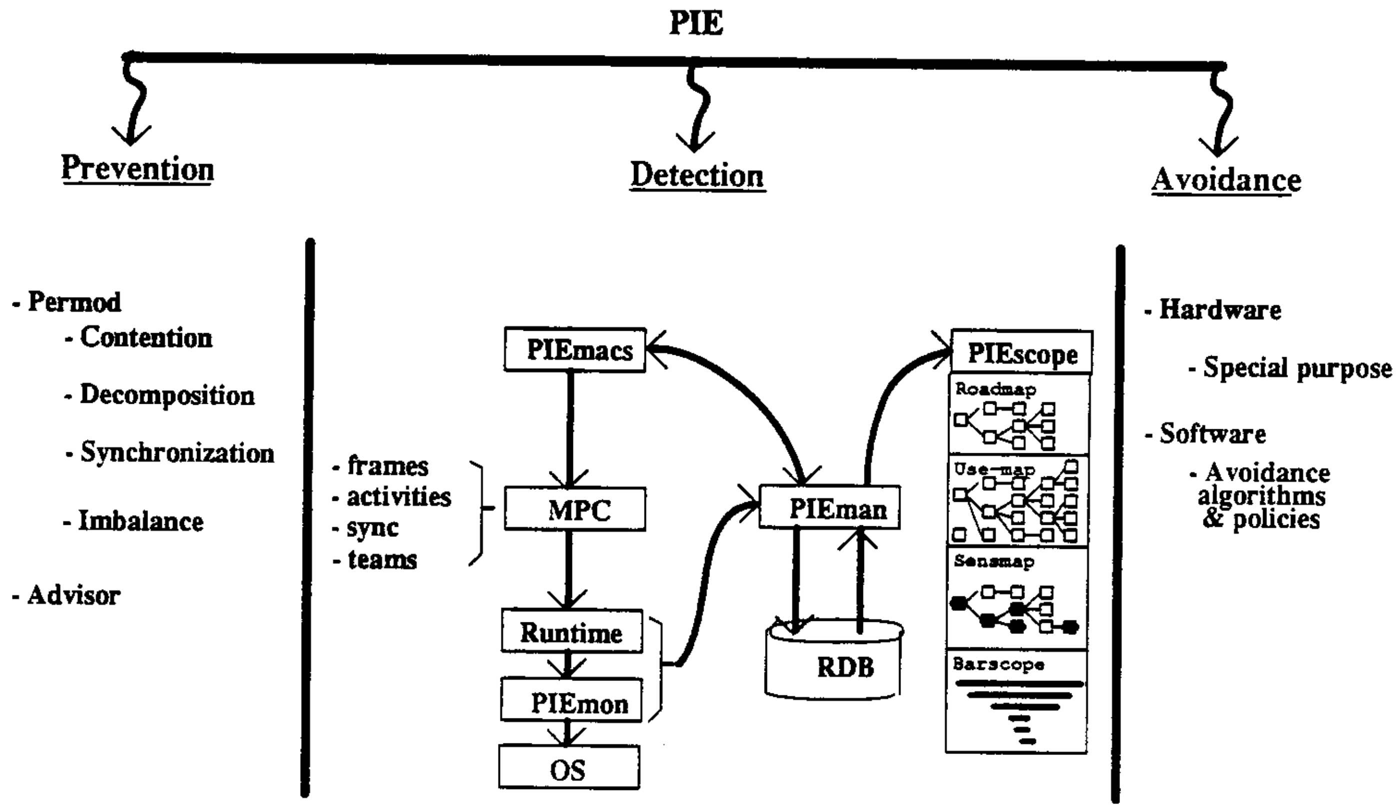

Figure 1-1: PIE -- Performance Efficient Parallel/Distributed Programming Environment

structure. During testing and debugging, the PIE system allows the user to view the execution of the program (either in-line or post-mortem). It is hoped the the extra information gleaned from the visual displays will help the user think more clearly about how his program's is behavior.

The present PIE environment consists of several components:

- PERMOD [Vrsalovic 84] is the performance modeling tool which allows for performance prediction in the early stages of the design of parallel systems.
- MPC (Multiprocessor C) is a C preprocessor that converts special MP (Multi-Processor)language constructs to C program syntax. It implements the "Consistent abstract shared data implementation machine" (CASDIM). Despite the fact that the target machine could be of a different architectures, MPC provides the CASDIM model to the user through the run-time resolution of the data consistency problems, and the physical synchronization, and communication.
- PIEman implements a relational model for each PIE IM. All PIE tools share data via the relational model.
- PIEmacs is a Gnu-Emacs based editor which extracts the development time data about the target program and assists in instrumenting it for the run-time monitoring purposes.
- PIEscope allows all the development and the run-time data to be presented to the PIE user in a graphical form.
- PIEmon supports the collection and storage of run-time events via the use of sensors.

The following text deals with the MPC part of the PIE only. Multiple examples are given throughout the text for illustrative purposes. A special section on how to use the MPC is added. The appendices contain the full MPC grammar as well as a number of MPC test programs.

2. MPC

Chapter 1 introduced the concept of the implementation machine or, IM. Unlike the typical virtual machine approach which relies on very generalized, high level interfaces which are reflected in the run-time structure of the code, the *implementation machine* approach translates the user code into target machine code using only low level calls to the run-time system. The current version of MPC supports the *Consistent Abstract Shared Datatype Implementation Machine* or, CASDIM {*see chapter*}.

MPC is a special preprocessor which translates MP syntax into a C program. It consists of three distinct parts: an analyzer, a constructor, and a target code generator.

The analyzer takes an MPC program as input which the constructor then converts to a C program. Although the resulting C program may differ from machine to machine, the original MPC program need not be changed. The analyzer also assists in instrumentation of the MPC program so that run-time performance data can be collected. In the present implementation, the target code generator is the C compiler. In the linking stage of the C compiler the user should use the mpc runtime support library.

The MPC language is modeled directly on C allowing parallel processing application programmers to use a language with which they are already familiar. All standard C commands and constructs are recognized by MPC. Identifiers, however, cannot begin with `mp_ or MP_`, since the constructor uses these as prefixes for internal identifiers. Consequently, virtually any program (noting the above mentioned exception) that compiles under C, will also compile under MPC. MPC merely adds several new constructs that allow for efficient parallel algorithm design, including:

1. **ACTIVITIES**: Sequential units of computation that are spawned and executed in parallel with the creating function.
2. **JOIN AND DETACH STATEMENTS**: Commands that allow activity management.
3. **FRAMES**: An encapsulation of global data and operations on that data. Frames are shared among specified activities and/or C functions and thus represent shared abstract data types.
4. **SYNC AND DSYNC STATEMENTS**: Meta constructs that provide for synchronization of parallel activities and used in frames to assure exclusion on the specific parts of the frame data.
5. **TEAMS**: Groups of activities and frames composing a unique subsystem with an associated communication and synchronization structure.
6. **SENSORS**: Location for collecting information on parallel program execution during run time.

The complete MPC grammar is included in Appendix XII and is a modified version of the C grammar in *A C Reference Manual* [Harbison and Steele 84] with the above MPC constructs added.

2.1. Activities

Parallelism is achieved through the use of *activities*. An activity is basically a procedure whose invocation spawns off another thread of control to execute the body of that activity in parallel with the calling activity.[1] MPC also provides constructs for joining with and detaching from activities.

2.1.1. Activity usage

An activity may be *declared* anywhere a data declaration is legal except inside structures and formal parameter declarations. A definition of an activity (or vector thereof) may be *instantiated* anywhere a data declaration is legal (even in structure and parameter declarations). An instance of an activity may be *invoked* anywhere a function invocation is legal (except in data initialization). Declarations and instantiations of activities are treated as data declarations and hence share a name space with normal C declarations. One might ask at this point why it is necessary to *instantiate* activities if no extra information is supplied at the time of instantiation. It is because both the user and MPC need a name for a particular activity when joining with or detaching from it (*see below*).

As was stated, activities are very much like functions. The differences being that they run in parallel to the calling activity and they do not return a value. Additionally, since an activity may run in a different process (depending on the architecture of the target machine), arguments are passed ***BY VALUE ONLY***. This means that passing pointers to an activity is not possible and, perhaps, not meaningful. The only way to share data between different activities is via the use of *frames* (see section 2.2), a construct for specifying abstract shared data types.

There are many applications in which the programmer would like to wait at a certain point for an activity (or set of activities) to finish executing before proceeding. MPC provides the ***join*** statement for such situations. ***Join*** appears as a function call that takes a list of instances of activities as it arguments. By default, upon completing, an invoked activity will wait until it is joined. This is undesirable if no activity will join with this one as it will continue to hold resources. If you know that no activity will attempt to join with the completed activity, you should include a ***detach*** statement in your code after invoking the activity. ***Detach*** appears as a function call and takes a list of activity instances as its arguments. Each activity instance passed to ***detach*** will exit immediately after performing its task.

Below we describe the syntax of activity related constructs in MPC and then present a simple application: parallel matrix multiplication. Note that all syntax specifications are given in BNF.

Activity *declarations* appear as follows (note that one may declare instantiations of an activity as part of the declaration of that activity):

activity-spec ::= *activity-tag-dcltr* { *parameter-dec* }*
compound-stmt

---

[1] Note that the procedure *main* is also considered an activity.

{ *activity-dcltr* #',' }* ';'

activity-tag-dcltr ::= `act` *identifier* '(' { *formal-dec* #',' }* ')'

An activity instance (or vector thereof) is of the form:

activity-dcltr ::= *identifier* { '[' *list-expression* ']' }*

and may appear as above (as part of an <activity-spec>) or in a semicolon-terminated list following the name of an activity declaration (as given in <activity-tag-dcltr>).

As stated above, *join* and *detach* appear as function calls. Their syntax is:

join-statement ::= `join` '(' { *activity-instance* #',' }* ');'

detach-statement ::= `detach` '(' { *activity-instance* #',' }* ');'

2.1.2. Example: Matrix Multiplication

The following example is an activity definition which includes instantiations of itself, and calls to the instantiations.

This activity performs parallel matrix multiplication. It achieves this by dividing up a large matrix into smaller subsections, and spawning activities that further divide the subsections, and then finally performs multiplication for some terminal subsection, combining results as each subtask finishes. For now, let us assume the existence of three shared matrices (the two factors and the product) that are global to the application. Also, assume that the function do_mult performs the actual multiplication for some *terminal* subsection of the matrix.

In Figure 2.1, the line defines this activity with the name `multiply`. It also shows that activities of this type need seven integer parameters. In the data declaration section of the activity, several local variables are declared, as well as two activities of type `multiply` called `subtask[0]` and `subtask[1]`. This is a good example of how activities can include instantiations of themselves. Self instantiation is also possible in frames and can be directly related to the way this can be done with structures.

The activity is passed two variables, `mx` and `my` representing limits placed on the granularity of how the matrix can be divided up. The parameters `x1`, `x2`, `y1` and `y2` define the submatrix that the activity has to work with. Several `if` statements check to see if it is still possible to divide up the submatrix further, and if so, the submatrix is divided in half and passed to two subactivities `subtask[0]` and `subtask[1]`, instantiated at the top of the activity, which in turn perform the same tests. If the submatrix cannot be divided any further (i.e. the dimensions are less than or equal to `mx` by `my`), the `do_mult` routine is called to calculate the product of the resultant matrix delimited by `x1`, `x2`, `y1`, and `y2`.

```
act multiply(x1,x2,y1,y2,mx,my,sz)
/***

    x1   x2
  y2******
    *    *
    *    *
  y1******

***/
int x1,x2,y1,y2,mx,my,sz;          /*sz is the original matrix size*/
{                                  /*mx and my are desired submatrix*/
  int ex,ey,i,j,k;                 /*dimensions*/
  float t,tmp;
  multiply subtask[2];             /* this is an instantiation of two
                                      activities of the same type */
  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;
  /* try to cut the longer side if possible */
  if (ex > ey){
    if (ex > mx) {
      /* cut along x dim. and give halves to children */
      subtask[0](x1,(x1 + ex/2 - 1),y1,y2,mx,my,sz);
      subtask[1]((x1 + ex/2),x2,y1,y2,mx,my,sz);
      join(subtask[0],subtask[1]);
      exit();
    }
    else if (ey > my) {
      /* cut along y dim. and give halves to children */
      subtask[0](x1,x2,y1,(y1 + ey/2 - 1),my,my,sz);
      subtask[1](x1,x2,(y1 + ey/2),y2,mx,my,sz);
      join(subtask[0],subtask[1]);
      exit();
    }
  }
  else if (ey >= ex) {
    if (ey > my) {
      /* cut along x dim. and give halves to children */
      subtask[0](x1,x2,y1,(y1 + ey/2 - 1),my,my,sz);
      subtask[1](x1,x2,(y1 + ey/2),y2,mx,my,sz);
      join(subtask[0],subtask[1]);
      exit();
    }
    else if (ex > mx) {
      /* cut along y dim. and give halves to children */
      subtask[0](x1,(x1 + ex/2 - 1),y1,y2,mx,my,sz);
      subtask[1]((x1 + ex/2),x2,y1,y2,mx,my,sz);
      join(subtask[0],subtask[1]);
      exit();
    }
  }
  else {
    /* no more children - do it ! */
    do_mult(x1, x2, y1, y2, sz);
  }
};
```

Figure 2-1: Parallel matrix multiplication

2.1.3. Hints on using activities efficiently

There are many ways to start a number of parallel activities, but one would always like to do this as efficiently as possible . The same is true for multiple join operations. The importance of this issue depends on the granularity of parallelism in the particular application.

The simplest way of spawning (and joining) N parallel activities is to use a loop construct like in the following example:

```
some_act my_activity[N];

for (i=0;i < N;i++) {
  my_activity[i](p1,p2,....,pN);
}

for (i=0;i < N;i++) {
  join(my_activity[i]);
}
```

It is often very useful to pass the index `i` as the parameter to the activity so their functionality can vary at the runtime.

A much more performance-efficient way to start up N parallel activities is to use recursions. In order to accomplish recursion the activity must include at least one instance of itself. The following example shows how one could start up a *pipeline* of activities using recursion.

```
act foo(....)
......
{
  foo bar;

  ..........
  do_some_work();
  ..........
  bar(.....);          /*  This will start a new
                          activity of type foo. */
  ........
  do_some_work();
  ........
  join(bar);
};
```

Better efficiency of such a solution comes from the fact that after the first activity is spawned in parallel it can start some more activities itself and thus the whole startup process could be done in parallel.

Keeping this in mind, let us revisit the matrix multiplication example (in Figure 2-1). The most natural way to use recursion when starting the activities was to form a binary tree where each activity starts two children and then waits for them to finish. However, closer examination reveals that when the startup procedure is done there will be n/2 - 1 parent activities waiting for children to finish some processing and n/2 activities doing actual useful work. Due to the fact that waiting activities consume system resources (despite the fact that they are blocked most of the time) there is a better scheme to start up n/2 activities.

This scheme is based on the fact that each activity cuts the work in half but passes only one half to the child while retaining the other half for itself. Thus, only working activities will exist, even though some of the activities will also have responsibilities to spawn and join children. Thus, the matrix multiplication activity from the previous example should be rewritten as follows:

```
act multiply(x1,x2,y1,y2,mx,my,sz)
int x1,x2,y1,y2,mx,my,sz;
{
  funct_multiply(x1,x2,y1,y2,mx,my,sz);
};
```

where `funct_multiply` is defined as follows:

```
funct_multiply(x1,x2,y1,y2,mx,my,sz)
    int x1,x2,y1,y2,mx,my,sz;
{
  int ex,ey,i,j,k;
  float t,tmp;
  multiply subtask; /*  This is an instantiation of
                        one activity of the same type. */
  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;
  /* try to cut the longer side if possible */
  if(ex > ey){

    if(ex > mx) {
      /* cut along x dim. and give a half to a child */
      subtask(x1,(x1 + ex/2 - 1),y1,y2,mx,my,sz);
      /* preserve half for yourself */
      funct_multiply((x1 + ex/2),x2,y1,y2,mx,my,sz);
      join(subtask);
      exit();
    }

    ........ as before ......

  }
}
```

Consider the time-lines generated by PIEscope in Figure 2-2. Both versions of matrix multiplication were executed using identical data. Note the difference in both the time it took to execute the algorithm as well as the number of activities used to calculate the product. The single child version of the algorithm was more efficient in resource utilization.

2.2. Frames

Since activities may execute in different address spaces, some mechanism is required for communicating between them. *Frames* provide a means for the programmer to specify and manipulate shared data objects. Basically, a frame is a collection of sharable data along with the operations that manipulate that data. For example, a frame could be composed of a data structure for a *queue* along

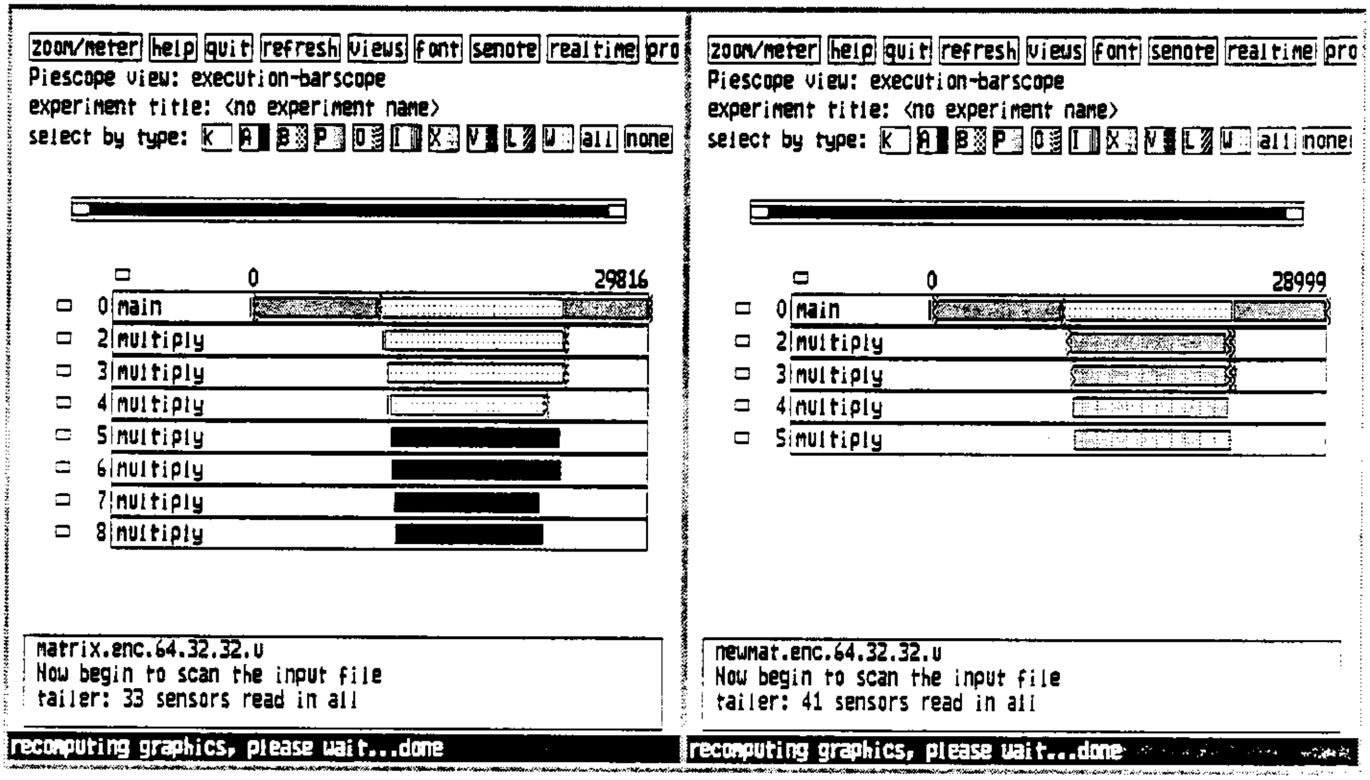

Figure 2-2: Time lines for two versions of matrix multiply

with the operations *put* and *get*. MPC also provides Synchronization support for these operations (both data and control-flow Synchronization is available).

2.2.1. Frame Usage

As with activities, frames may be *declared* anywhere a data declaration is legal (expect in structures and formal parameter declarations). A frame declaration is really a template which takes arguments. These arguments are usable as constants when defining the global data and operation of the frame. The user supplies the arguments to the template at instantiation time. A frame *instantiation* may appear anywhere a data declaration is legal (including in structures and parameter declarations). Just as with activities, frame *declaration* and *instantiation* names share the standard C name space.

As stated above, *frames* are an encapsulation of a data object for use by parallel access. Thus, the first thing defined in a frame[2] is the data it encapsulates along with any internally used declarations. The frame local data can be of *any* legal C type as well as declarations and/or instantiations of other frames and activities.

After the data encapsulated by the frame (and any internal data) has been declared, the operations on that data must be defined. Operations are implemented as in-line functions. When a call to an operation is seen by MPC, any parameters passed to an operation are substituted into the operation definition and

[2]See section for a description of the frame syntax

the code is expanded in-line by the code generator. Note that local data declarations within an operation is permissible.

Operations that return a value require *exactly one* `export` statement somewhere in their body. The export statement is analogous to the return statement in that it specifies the value to be returned by the operation. However, the export statement *does not branch out of the operation.* All commands before the `export` in the operation definition are expanded before the statement that includes the call to the operation. All commands after the `export` are expanded after the calling statement. The expression within the `export` is ***expanded directly into the calling line.***

The semantics of the export statement has some serious ramifications on the definition and usage of frame operations. For one, since only one export statement can appear in the code, the user should create a local variable for containing the result if the result of the operation could be generated in one of several branches of a condition. In addition, it means that they can be unfolded as the LHS or RHS of expressions only. That is, export statements cannot appear as arguments to procedures or in conditional clauses. To make this latter problem more clear let us consider the following example:

```
opr int test()
{
  int a;

  a =(read_ptr < write_ptr);
  export(a);
}
```

To use this frame operation as the test for a while loop, code the loop as follows:

```
temp = frame.test();
while(temp){
  .
  .
  temp = frame.test();
}
```

which is unfolded into C code like:

```
{
  int mp_xx_a;

  mp_xx_a =(int)(frame[0]->read_ptr < frame[0]->write_ptr);
  temp = mp_xx_a;
  while(temp){
    .
    .
    mp_xx_a =(int)(frame[0]->read_ptr < frame[0]->write_ptr);
    temp = mp_xx_a;
  }
}
```

In future versions of the MPC compiler, substitution of the local variables will be done automatically. Thus, frame operations will be permissible in almost all contexts.

Finally, frames must also contain some initializing function at the end of their definition. This function can be null, but open and close braces must be present. Every time a frame is instantiated the initialization function for that frame is executed. A common use for the initialization function is the initialization of global memory. An example of usage of the initialization function is provided later.

2.2.2. Frame Syntax

The syntax related to frames is described below. Following that is a detailed description of how the ***sync*** and ***dsync*** statements is given. Finally, several examples are presented on how one might implement and use different common data types.

Frame *declarations* appear as follows (note that, as with activities, one may declare instances of a frame as part of its declaration). All syntax forms are given in BNF.

```
frame-definition ::= frame-spec { frame-dcltr #',' }* ';'

frame-spec ::= frame-tag-dcltr { parameter-dec }* '{' frame-dec '}' ';'

frame-tag-dcltr ::= frame identifier '(' { formal-dec #',' }* ')'

frame-dec ::= { local-data-dec }* { frame-operation }* frame-initialization

frame-operation ::= opr { type-class-spec }* operation-name { parameter-dec }*
                          operation-body

operation-name ::= identifier '(' { formal-dec #',' }* ')'

operation-body ::= '{' local-data-dec
                          { statement }*
                          export-statement?
                          { statement }* '}'

export-statement ::= export '(' list-expression ');'

frame-initialization ::= compound-stmt

sync-statement ::= sync '(' { opr-name #',' }* ')'
                          compound-stmt
dsync-statement ::= dsync '(' list-expression ')'
                          compound-stmt
```

The syntax for instantiating a frame (or vector thereof) is:

```
frame-dcltr ::= identifier '(' list-expression ')' { '[' list-expression ']' }*
```

and may appear within the declaration of a frame or in a semicolon-terminated list following the name of a frame (as given in *<frame-tag-dcltr>*).

An invocation of a frame operation is of the form:

```
frame-opr-call ::= frame-instance '.' opr-name '(' list-expression ')'
```

```
where,
     frame-instance ::= identifier { '[' list-expression ']' }*

   frame matrix(rank)
       int rank;

   {
     float mat[rank][rank];

     opr float get(i,j)
         int i, j;
     {
       export(mat[i][j]);
     }

     opr float put(i,j)
         int i, j;
     {
       export(mat[i][j]);
     }

     {
       bzero(mat, sizeof(float)*rank*rank);
     }
   } a(5), b(5)[5][5];
```

Figure 2-3: A simple frame

One example of a frame definition is given in Fig. 2-3. This frame implements a matrix whose elements are floating point numbers. The rank of the matrix is specified at instantiation time. In this example, `a` is an *instance* of a 5x5 matrix and `b` is a 5x5 vector of 5x5 matrices (i.e. b consists of 25 *separate* instances of a 5x5 matrix).

2.2.3. Synchronization within a frame

The above example is fine when you know that the users of a particular instance[3] of the frame will never be using it at the same time. In most applications however, this is not the case. One client may be modifying a cell while another is looking at the value of that cell. This is clearly undesirable. Thus, some sort of mutual exclusion must be specified on the data and operation of a frame.

The sync and dsync statements allow synchronization of frame operation parts that are performed in parallel. In other words, since frame operations perform actions on shared memory, sync and dsync statements provide for mutual exclusion of access to parts of frame memory used by parallel activities.

[3]It is important that the reader recognize the fact that in a vector of frames instances (as with vectors of activity instances), the components of the vector are not related in any way other than that they share the same definition.

2.2.3.1. Sync

Sync statements can be included only inside the definition of a frame operation. A sync statement precedes a block of critical code that begins with the sync statement and ends at the end of the sync reach (i.e. at the closing brace). The sync statement contains a parenthesized list of names of operations which also have critical sections of code that may not be executed while the code in the block is executed. To execute a sync statement is to perform synchronization on the frame operations named in the parameter list. A frame operation can only perform synchronization on itself or on other frame operations within the same frame. If a frame operation is named in the parameter list, that operation must also have a sync statement which precedes its own critical section (if an operation named in the sync parameter list does not have a sync statement, it should not have been named in the list). When an activity executing a frame operation, `a`, performs synchronization on another frame operation, `b`, a condition (transparent to the programmer) is set which causes any other activity executing the sync statement in `b` to block until the activity executing `a` exits its synchronized block of code.

```
frame dummy()
{
  opr a()
  {
    ...
    sync(b)  {
      ...
    }
    ...
  }

  opr b()
  {
    ...
    sync()  {
      ...
    }
    ...
  }

  {}
}
```

Figure 2-4: Frame skeleton with sync statements

Let's examine some hypothetical cases using the frame shown in Figure 2-4. First, let's assume that there are only two activities, `A` and `B`. Let `A` call operation `a()` and `B` call `b()`. If `B` executes the `sync()` statement both *after* `A` has executed the `sync(b)` statement and *while* `A` is still executing the braced code following `sync(b)`, `B` will block until `A` exits that code. The empty sync statement in `b()` means that although `b()` is not synchronizing on any other operations, other operations may synchronize on it.

For the second example, let's assume that we still have only two activities, `A` and `B`. Again, let `A` call operation `a()` and `B` call `b()`. If `B` executes `sync()` *before* `A` has executed `sync(b)`, `A` will *not* block when it executes `sync(b)` even if `B` has not exited the critical section in `b()`. This is because the sync statement in `b()` does not contain the name of `a()` in its parameter list. In addition, even if `A` executes `sync(b)` while `B` is in the critical section of `b()`, `B` does not block. Although it may seem that the synchronization protocol shown in Figure 2-4 has no viable application, it illustrates the behavior of the MPC sync statements.

In this manner, parts of operations which would conflict in some way with parts of other operations can be made to be mutually exclusive. One should note that `sync` statements are just the first step to higher synchronization constructs based on path expressions and thus will be automatically generated in future versions of MPC preprocessors.

2.2.3.2. Dsync

The Sync statement allows for synchronization of arbitrary control points in frame operations executed in parallel regardless of which part of the frame's global data these operations are accessing. The dsync statement allows for synchronization of accesses to particular data items. Like sync statements, they are only allowed inside operations. Dsync takes as parameters a list of frame variables, separated by commas, which are to be exclusively used. When a part of an operation within the reach(i.e. braces) of a dsync statement is executed, if any of the variables in the statement have already been protected by another dsync, the activity will have to stop and wait for the execution of the other operation to finish. This command is used when certain frame variables are being changed by an operation and it is desired that no other activity touch the variables until the changes have been completed. Matrix variables can have the expressions which will be evaluated at the runtime as their indices (ie. dsync(a[i]) is legal if i will be calculated at runtime prior to the time dsync is executed).

2.2.3.3. Synchronization example

Let us return to the example in Figure 2-3 to see how the contention problem might be solved using the MPC Synchronization constructs. What we have to watch out for is two parallel activities either writing at the same time or reading and writing at the same time. Thus activities reading is not a problem as reading is not a destructive operation. So, a straight forward approach might produce something like the following code:

This code segment does what we specified above however, a less superficial look at the problem shows us that the *granularity* of the above Synchronization is quite coarse. No matter what cell a client is writing to, no other client may read or write to another cell. What is really desired is mutual exclusion, not on a *per-operation* basis, but on a *per-cell* basis. That is, in this case, Synchronization on the basis of *data* is more efficient than synchronizing on the basis of *control flow*. Thus, a more efficient solution might be:

There are situations which require more than one data element to be used atomically at the same time. In such cases a list of data elements can be given to dsync, which will then employ a deadlock avoidance algorithm to lock atomically all the elements in the list. One should be very careful not to use nested sync and dsync statements due to the fact that this can lead to the potential deadlock situations.

```
frame matrix(rank)
    int rank;

{
  float mat[rank][rank];

  opr float get(i,j)
      int i, j;
  {
    sync (put) {
      export(mat[i][j]);
    }
  }

  opr float put(i,j)
      int i, j;
  {
    sync (put, get) {
      export(mat[i][j]);
    }
  }

  {
    bzero(mat, sizeof(float)*rank*rank);
  }
};
```

Figure 2-5: A MPC implementation of a shared matrix

```
frame matrix(rank)
    int rank;

{
  float mat[rank][rank];

  opr float get(i,j)
      int i, j;
  {
    dsync (mat[i][j]) {
      export(mat[i][j]);
    }
  }

  opr float put(i,j)
      int i, j;
  {
    dsync (mat[i][j]) {
      export(mat[i][j]);
    }
  }

  {
    bzero(mat, sizeof(float)*rank*rank);
  }
};
```

Figure 2-6: Matrix with more efficient synchronization

2.2.4. Frame Examples

2.2.4.1. Stream

The shared stream is a basic object used in many distributed applications. One possible definition based on a circular queue is given in Figure 2-7.

```
frame stream (length)
int length;
{
  char stream_data[length];
  char *read_ptr,*write_ptr;

  opr int get(c)
      char c;
  {
    int stream_is_empty;

    dsync (read_ptr, write_ptr) {
      stream_is_empty = (read_ptr == write_ptr);
      if (stream_is_empty) result = 1;
      else {
        if (read_ptr >= (stream_data + length)) read_ptr = stream_data;
        c = (*read_ptr++);
      }
    }
    export(stream_is_empty);
  }
  opr int put(c)
      char c;
  {
    int stream_is_full;

    dsync (write_ptr, read_ptr) {
      stream_is_full = ((long)read_ptr) - ((long)write_ptr);
      stream_is_full = (stream_is_full == 1) ||
                          (stream_is_full == -length);
      if (stream_is_full) result = 1;
      else {
        if(write_ptr >= (stream_data + length)) write_ptr = stream_data;
        (*write_ptr++) = c;
      }
      export(stream_is_full);
    }
  }
  {
    write_ptr = (read_ptr = stream_data);
  }
};
```

Figure 2-7: A mpc implementation of a byte stream

Any instantiations of this frame requires one integer parameter which specifies the length of the stream. Within the body of the `stream` frame is its global data, `stream_data`, which is the array storing the que values, and `read_ptr` and `write_ptr`. The pointers serve to mark the beginning and end of the stream. The frame has two operations defined: `get`, and `put`. `get` returns the character which is on top of the stream, pointed to by `read_ptr`. The operation should be used with a command such as:

```
xxx.get(yy);
```

where yy is a variable of type `char` (because `get` is of type `char`), and xxx is the name of an instance of a frame of type `stream`. This operation will return non-zero if the stream was empty. The second operation, `put`, is used to store data in the stream. It is used with the form:

```
xxx.put(yy);
```

where xxx and yy are defined as before. It will return non-zero if the stream was full.

2.2.4.2. Mailbox

Once the frame `stream` is defined, it can be used as an abstract data type to instantiate a shared stream in a MPC program. The following program example uses the basic queue definition to build a mailbox to illustrate how frames can be nested. If a frame definition is local to a specific frame it can be treated in the same way data structures are treated in C with respect to local declarations. The fact that frame `list` is defined inside frame `mailbox` means that other frames with the same type name (ie. `list`) can be defined in parallel branches of the same program. In the following example the frame `list` is local to the frame `mailbox` and other definitions for `list` can coexist within the same program.

The parameters passed to this frame when instantiated specify that `CUSTOMER_NUMBER` frames of type `queue` (defined above), each with a size of `CUSTOMER_SIZE` bytes should be defined local to that instance. Besides having several frames of type `que` in its global data space, mailbox frames also have a frame of type `list`, which is actually defined within this frame's definition. This means that `list` type frames can only be instantiated within `mailbox` type frames, and that there could exist different frame declarations of a type `list` outside this scope. The `list` frame has as its global data an array of strings. It has an operation, `find`, which searches the array for a particular name, and an operation, `enlist`, which copies a name into the array. The initializing procedure clears the strings in the list to NULLs. Trailing the definition of the frame, is a declaration for a frame of this type called `mailbox_name_list`.

Next begin the operations for the frame defined as `mailbox`. The first of these is called `send`, and it is used to put a string into the que frame of the receiver. Names of customers are stored in the list of names inside `mail_name_list` with the operation `allocate`. `deallocate`, does the reverse, and is used to clear the list. The operation `locate` searches the list for a particular name. Finally, `read` checks the user's que, and if it is not empty, grabs its contents and puts it into a buffer. No initializing procedure is necessary, and there are no trailing instantiations. If one wanted to instantiate a frame of this type (type `mailbox`), one could include the statement:

```
mailbox mbl(1);
```

This would create one frame of this type and run the initializing procedures of the frames of types `que` and `mail_name_list` included within frames of type `mailbox`.

As in Pascal scoping, only frames of type `mailbox` can see the definition for the frame type `mail_name_list`, and therefore are the only places where this type of frame can be instantiated.

```
frame mailbox(customer_number)
int customer_number;
{
  stream mailbox_que(CUSTOMER_SIZE)[customer_number];
  frame list(list_size)/* here starts the internal */
    int list_size;      /* frame definition */
  {
    struct {
      char names[NAME_LEN];
    } name_list[list_size];

    opr int find(name)
      char *name;
    {
      int ret_value, i;

      ret_value = UNSUCC;
      for(i=0;i<list_size;i++){
        if(strcmp(name_list[i].names,name) == MATCH){
          ret_value = i;
          break;
        }
      }
      export(ret_value);
    }

    opr enlist(name,id)
      char *name;
    int id;
    {
      export(strcpy(name_list[id].names,name));
    }
    {
      int i;

      for(i=0;i<list_size;i++)
        name_list[i].names[0]='\0';
    }
  }mailbox_name_list(CUSTOMER_NUMBER); /*this is the instantiation*/
```

Figure 2-8: A MPC implementation of a mailbox -- first part

```
opr int send(id,buff,len)
    int id;
    char *buff;
    int len;
{
  int i;
  register int temp;

  for(i=0;i<len;i++) {
      temp = mailbox_que[id].put(*(buff+i));
      if (temp) {
        printf("sender queue full\n");
        break;
      }
    }
  }
  export(i);
}

opr int allocate(customer_name)
  char *customer_name;
{
  int ret_value, id;

  sync (allocate) {/* NO OTHER ALLOCATIONS In PARALLEL */
    ret_value = mailbox_name_list.find(customer_name);
    if(ret_value == UNSUCC){
      id = mailbox_name_list.find("");
      if(id != UNSUCC){
        mailbox_name_list.enlist(customer_name,id);
        ret_value = id;
      } else  ret_value =UNSUCC;
    } else ret_value = SUCC;
  }
  export(ret_value);
}
opr int deallocate(id)
  int id;
{
  export(mailbox_name_list.enlist("",id));
}
```

Figure 2-7, continued

```
  opr int locate(name)
    char *name;
  {
    export(mailbox_name_list.find(name));
  }

  opr int read(id,buff,len)
    int id;
    char *buff;
    int len;
  {
    int i;
    register temp;
    i = 0;
    while (i < len) {
      temp = mailbox_que[id].get(buff[i]);
      if (temp == 0)
        if (buff[i++] == '\0') break;
    }
    buff[len-1] ='\0';
    export(i);
  }
  {/* Lists and queues are already initialized */ }
};
```

Figure 2-7, continued

2.2.4.3. Dynamic Frames

Frames behave like static variables (ie. at the moment of instantiation all the memory for the frame data is allocated). There are many situations where the programmer does not know ahead of time how much memory is needed in each frame. In such a case one can use the different strategy of specifying only a pointer to global data, which can then be allocated at run-time.

The following example shows how to use frames which allow for dynamic global memory usage. The first frame is a simple global heap and the second one is a memory allocator which uses the heap to refill the buckets each time any of them becomes empty. Both frames use the synchronization command `dsync`.

```
frame HEAP_TYPE(init_size)
int init_size;
{
  char *heap_ptr;
  int heap_size;

  opr char *get(size)
    int size;
  {
    char * temp;

    dsync(heap_ptr){
      if(heap_size > size) {
        temp = heap_ptr;
        heap_ptr += size;
        heap_size -= size;
      }
      else {
        if ((heap_ptr = mp_alloc(init_size)) == NULL)
          PANIC("NO MORE MEMORY!");
        temp = heap_ptr;
        heap_ptr += size;
        heap_size = init_size - size;
      }
    }
    export(temp);
  }

  {
    if ((heap_ptr = mp_alloc(init_size)) == NULL)
      PANIC("NO MORE MEMORY!");
    heap_size = init_size;
  }
};
```

Figure 2-9: A MPC implementation for a shared global heap

The heap in Figure 2-9 serves as a buffer between the user and the operating system. Mp_alloc calls are issued only when the previous block on a heap is exhausted or the remainder is smaller than the size needed. It can be seen that only the heap pointer and the heap size are defined as global frame data. The heap memory itself will be allocated first at instantiation time. It will also be allocated any time the heap block becomes too small to accommodate a new requested size.

Using the heap definition as given above, one could write a more complex frame which can be used as a dynamic memory allocator in application space. This frame definition uses the standard "bucket" allocator scheme, where each bucket holds memory blocks of the size 2^i and i is the bucket index.

Block size is always adjusted to the nearest greater 2^i and taken from the corresponding bucket. If the bucket in question is empty a new block of size 2^i will be allocated from the heap.

After they are used, blocks are returned to a bucket holding a list of unused blocks of size 2^i. There are two operation exported to users:

- Allocate: Allocate a global chunk of <size> and return the pointer to it. Actually it will always return pointer to the chunk start + sizeof(integer) due to the fact that chunk is going to keep the index of its bucket in size field, and next is going to be overwritten.
- Free: Free the chunk pointed to by base. To reclaim the whole chunk one should decrease base by the size of the size field

An MPC implementation of a simple global memory manager is presented in Fig.2-10. MP_MEM type is defined as follows:

```
#typedef struct mp_mem{
    int                     size;
    union {
        char            *c;
        struct mp_mem   *m;
    }                       next;
}MP_MEM;
```

One should note that in this particular example all buckets are filled with exactly one chunk of a size 2^i at the instantiation time. If needed the user can decide otherwise by redesigning the initialization section.

2.2.5. Queues

The Queues previously described were very simple circular queues having a constant element size and number. The next example shows how to implement a more general shared FIFO queue. It is assumed that the elements to be put in the queue are of various sizes and that the first member of a structure representing an element is the pointer to an element of the same size.

There are two operations in the queue definition: `push` and `pop`, which are self explanatory. Due to the fact that both operations use the head pointer of the queue. Dsync statements define the critical sections. An example queue is presented in Figure 2-11. In this particular example queue is initialized to be empty (i.e. head and tail are set to NOITEM).

2.2.6. Semaphores

The PIE environment encourages the use of the higher level synchronization commands which employ "blocked - wait" primitives at runtime. However if there is a need for explicit control flow synchronization, a semaphore can be built as shown in Figure 2-12. This semaphore supports the following operations:

- Wait: Wait takes no arguments and it returns nothing. Will block caller, via mp_wait, if

```
frame memmgr(max_block_size)
     int max_block_size;
{
  HEAP_TYPE  heap(HEAP_SIZE);
  MP_MEM *buckets[log(max_block_size)/log(2)];

  opr char *memget(size)
    register int size; /* size in bytes */
  {
    register int i;
    register int tmp;
    register MP_MEM base;

    size = size + sizeof(int);
    if (size < sizeof(MP_MEM)) size = sizeof(MP_MEM);
    tmp = 1;
    for (i=0;tmp < size;i++) tmp = tmp<<1;
    dsync(buckets[i]){
      if (buckets[i].m != NOMEM) {
        base.m = buckets[i].m;
        buckets[i].m = base.m->next.m;
      } else {
        base.m = heap.get(tmp)
        base.m->size = i;
      }
    }
    export((base.c + sizeof(int)));
  }

  opr void memfree(base)
    register MP_MEM base;
  {
    register int n, tmp;
    if (base.m != NOMEM) {
      base.c = base.c - sizeof(int);
      n = base.m->size;
    }
    if (n >= 3 && n < (log(max_block_size)/log(2))) {
      dsync(buckets[n]){
        base.m->next.m = buckets[n].m;
        buckets[n].m = base.m;
      }
    } else PANIC("Size to free OUT OF RANGE!\n");
  }
  { /* this is executed upon initialization */
    int j,s;
    /* fill in the buckets */
    for (j=3,s=8;j<(log(max_block_size)/log(2));j++,s<<1){
      buckets[j].m = heap.get(s);
      buckets[j].m->next = NOMEM;
      buckets[j].m->size = j;
    }
  }
}
```

Figure 2-10: Memory manager implemented as a MPC frame

semaphore is less than one. Note: mp_wait will open the lock mutex before the calling process is put to sleep. So, despite what it looks like signal will be able to execute.

```
typedef struct xitem{
  struct xitem *next;
}*mp_xitem_p;

#define NOXITEM (mp_xitem_p)0

frame mp_xqueue()
{

  mp_xitem_p      head;
  mp_xitem_p      tail;

  opr void push(item)
    mp_xitem_p item;
  {
    dsync(head){
      my_act->next = NOXITEM;
      if (tail == NOXITEM) head = item;
      else tail->next = item;
      tail = item;
    }
  }

  opr mp_xitem_p pop()
    {
      mp_xitem_p        i;

      dsync(head){
        i = head;
        if ( i != NOXITEM)  {
          head = i->next;
          if (head == NOXITEM) tail = NOXITEM;
        }
      }
      export(i);
    }
  /* init */
  { head = tail = NOXITEM;}
};
```

Figure 2-11: MPC implementation of a queue

- Signal: Signal takes no arguments and returns nothing. Will wake first waiting processes in the wait queue.
- Signal_all: Signal takes no arguments and returns nothing. Will wake all waiting processes in the wait queue.

The frame in this example implements a FIFO semaphore having the following three operations:

wait	invocation is done in an activity by calling xxx.wait(). After this call activity is suspended until next signal_all/or signal command.
signal	(xxx.signal()) operation will wake up the activity from the top of the queue.
signal_all	(xxx.signal_all()) command will wake up all the activities from the semaphore's queue.

```
frame mp_semaphore(init_val)
int init_val;
{
  int s;
  mp_cond c;
  mp_lock mutex;/* lock for mutual exclusion of the two oprs */
  opr void wait() {
    mp_close(&mutex);
    if (s < 1) mp_wait(&c, &mutex);
    s--;
    mp_open(&mutex);
  }
  opr void signal() {
    mp_close(&mutex);
    s++;
    if (s > init_val) s = init_val;
    mp_signal_first(&c);
    mp_open(&mutex);
  }
  opr void signal_all() {
    mp_close(&mutex);
    s = init_val;
    mp_signal_all(&c);
    mp_open(&mutex);
  }
  {
    mp_lock_init(&mutex);
    mp_cond_init(&c);
    s = init_val;
  }
};
```

Figure 2-12: MPC implementation of a semaphore

2.2.7. Barriers

There is one kind of synchronization which cannot be easy realized by using syncs or dsyncs. This is the barrier synchronization in the case where there is no busy waiting allowed. MPC itself introduces one implicit kind of barrier synchronization i.e. the JOIN statement. However, in practical situations the JOIN may be too costly to perform. This may be due to the underlying system on top of which CASDIM is implemented. In such situations the user may wish to implement his own barrier synchronization. One way to do this in MPC is to given in Figure 2-13.

```
frame barrier(N)
int   N;  /*  N is the number of activities
              involved in the synchronization. */
{
  mp_xqueue Q;
  int counter;

  opr void block()
    {
      int tcount;
      mp_xitem_p   titem;

      dsync(counter)
        {
          tcount = counter++;
          Q.push(my_act);
          my_act->state = WAITING;
        }
      if (tcount == N)
        {
          counter = 0;
          titem = Q.pop();
          while (titem != NOITEM)
            {
              titem->state = RUNNING;
              titem = Q.pop();
            }
        } else {
          while(my_act->state == WAITING)
            mp_swtch();
        }
    }
  {  counter = 0; }
}
```

Figure 2-13: A barrier coded in MPC

3. Using MPC

As of this printing, the MPC compiler resides in the usr/pie/bin sub-directories on machines it is installed on. To use MPC, the following should be added to the path statements in the user's .login file:

```
setpath -ia $home /usr/pie
```

The syntax for compiling an mpc program is:

mpc *file*.mpc [{-#mpc options}] [{-cc options}]

The command line options are as follows:

-#p: This will force MPC analyzer to parse the MPC source even if the *.pif file is available and up to date.

-#cnn: The c is used for incremental compiling. When the source code is divided into several subfiles, for examples, with one containing global frames, one containing procedures and another containing main, each file can be compiled separately into a .o module which can later be linked with the other modules. User should supply a unique module number (nn) for each separate mpc file. In a case of separate compilation all frames definitions which are global to the entire program must be in include (*.h) files. They must also be instantiated in these files, and then be included in all the modules that use these frames and should also be visible (ie. included) in the main module. To link all the modules together, the C compiler should be called using the command:

cc *file1*.o *file2*.o -lmpc -lmach

-#C: If this option is included in the command, mpc will dump the file that is sent to the C compiler in the file *file*.c.

-#l: This option will make MPC print out what it is doing with the mpc file as it compiles.

-#m: If an 'm' is in the option, MPC will include the MPC monitor library which causes the print out of monitoring information when the compiled mpc program is executed.

-#d: A 'd' will cause MPC to include the mpc debugging library. At the start of execution of an MPC program compiled with this option, run time support will enter interactive debugger. For details see MPC debugger documentation, or type 'help' while in the debugger.

-#Ynn: Will set the lock yield count to nn (ie. after encountering the closed lock, each activity will spin nn times, and then suspend itself by calling swtch OS call). Default value for nn is 200. On the ENCORE, each spin takes about 20 microseconds.

-#Gnn: Will set the size of runtime global heap to nn Mbytes. Default value is 1Mbyte.

`-#Mnn`: Will set the size of compiler heap to nn Mbytes. Default value is machine and OS dependent.

`-#Pnn`: Will set the maximum number of user processes which can be created at the runtime to nn. Default value is 16.

`{cc options}`: Besides the '-#' switches that are directed at the mpc compiler, normal cc switches can be included on the command line, such as the inclusion of libraries, request for the assembler file, etc.

NOTE: Some make script interpreters accept '#' character as the comment delimiter regardless of its position in the line. For this reason mpc preprocessor accepts also "--" notation instead of "-#" notation to be used in front of mpc compile time switches.

4. Advanced MPC

This chapter discusses three MPC run-time libraries, a block timing facility and advanced MPC programming tricks. First, the standard run-time structures and functions are discussed. After these some advanced MPC programming tricks are presented. Finally, a MPC source template and the corresponding produced C code is given.

4.1. MPC Run-time Support: Standard Data Structures and Functions

There are three MPC run-time libraries, a standard or normal library, a performance monitoring library and a debugging library. The libraries contain the data structures, procedures and other entities used by MPC to support the execution of an MPC program. The user may want to call one or more of these run-time routines, although this is not recommended, especially in the case of those routines (and activities) used exclusively by either the monitoring or debugging libraries which will be described in the separate documents. One should consult with a PIE group member before using any of the calls or data structures described herein.

4.1.1. Standard MPC run-time structures

There is a set of standard structures used in all of the MPC run-time libraries. The members of this set are discussed below. In many cases there are some elements of a structure that only apply to the debugging library. Since the debugger is presently under development, these additional elements are not yet fully supported.

4.1.1.1. Naming

The mp_name type is defined as a pointer to a string of characters, and is the type used to store the names of queues, activities, and other MPC run-time objects.

```
/*** mp names ***/
typedef char     *mp_name;
```

4.1.1.2. Byte addressing

Pointers to absolute memory locations are type memory_p and are defined as pointers to characters.

```
/*** memory pointer ***/
typedef char    *memory_p;
```

4.1.1.3. Queues

Figure 4-1 shows the queue structures used by MPC. The mp_queue structure is implemented as a linked list. Objects to be queued into them must have a pointer to the same object as the first element in the data structure (eg. see mp_item). Queues, like all MPC objects, have provisions for a naming scheme if the DEBUG option is invoked.

```
typedef struct item {
        struct item     *next;
}               mp_item, *mp_item_p;

typedef struct queue {
        mp_item_p       head;
        mp_item_p       tail;
#ifdef  DEBUG
        mp_name         name;
#endif  DEBUG
}               mp_queue, *mp_queue_p;
```

Figure 4-1: Queue structures

4.1.1.4. Locks

Figure 4-2 shows the lock structure used by MPC. Locks are grabbed when exclusive access to memory is desired by a run-time library routine. When a routine attempts to grab a certain lock and that lock has already been locked, execution is suspended until the lock is released. This is done in two steps: First, blocked activity will spin on a lock for a nuber of spins defined by --Y switch. If, after the spining ended, the lock in question is still closed the blocked activity will suspend its execution.

```
typedef struct {
        int             lock;
#ifdef  DEBUG
        mp_name         name;
#endif  DEBUG
}               mp_lock, *mp_lock_p;
```

Figure 4-2: Lock structure

4.1.1.5. Conditions

Figure 4-3 shows the condition structure used by MPC. The `mp_cond` type is the MPC support for signaling that a condition has been met. It contains a queue and coresponding lock to ensure atomic queue operations.

```
typedef struct cnd {
        mp_lock         lock;
        mp_queue        queue;
#ifdef  DEBUG
        mp_name         name;
#endif  DEBUG
}               mp_cond, *mp_cond_p;
```

Figure 4-3: Condition structure

4.1.1.6. Synchronization

Figure 4-4 shows the synchronization structure used by MPC. All frame operations have a structure of type `mp_opr` associated with them. It used to support the MPC `sync` statement. The structure contains a variable, `count`, for indicating how many operations have performed a `sync` within arbitrary (but identical) frame operations. An operation is allowed to proceed pass a `sync` if and only if `count` is equal to zero. The second element in mp_opr is a condition variable to support blocked waiting. Like all MPC structures, mp_opr supports a naming scheme in DEBUG mode.

```
typedef struct opr {
        int             count;
        mp_cond         cond;
#ifdef  DEBUG
        mp_name         name;
#endif  DEBUG
}               mp_opr, *mp_opr_p;
```

Figure 4-4: Synchronization structure

4.1.1.7. Global memory management

Figure 4-5 shows the structures used by MPC for global memory management. mp_mem is a structure type used to grab a free block of memory from a global heap. MPC run-time implements "bucket" memory manager, where sizes of free blocks allocated from the buckets are equal to powers of 2.

```
typedef char    *mp_pointer;

typedef struct mem {
        int             size;
        union nn {
                struct mem      *m;
                mp_pointer      c;
        }next;
}               mp_mem, *mp_mem_p;

typedef union{
        char    *c;
        mp_mem_p m;
}memun;

#define K 1024
#define MEGABYTE K*K
#define NOMEM (mp_mem_p)0
#define NIL (mp_pointer)0
```

Figure 4-5: Global memory management structures

4.1.1.8. Activity control block

The type mp_acb is a structure, called an *activity control block* or *acb*, representing an activity in MPC. Figure 4-6 shows *acb* structure used by MPC. The run-time uses the structure to manage the creation (each activity get its own acb upon creation) , scheduling, lineage (pointers to parent and children), and termination of activities. A local pointer to each activity's activity control block is kept in my_act.

4.1.1.9. Workload control block

In addition to my_act, each activity knows about a global pointer to a workload control block, mp_wcb, which keeps the global parallel workload state. Figure 4-7 shows workload control block structure used by MPC.

```
typedef struct acb {
   struct acb      *next;          /*** for queueing purposes  ***/
   mp_lock          lock;          /***
                                    *** lock used when activity state,
                                    *** ie, the activity control
                                    *** block, is updated.
                                    ***/

   mp_cond          cond;          /***
                                    *** condition that is waited
                                    *** on while waiting for join
                                    ***/

   int              status;        /***
                                    *** what kind of activity am I?
                                    *** IDLE, MAIN, ACT, JOINED,
                                    *** DETACHED, DONE
                                    ***/

    int              state;        /*** IDLE, RUNNING, WAITING ***/
    int              id;           /*** pid of activity        ***/
    jmp_buf          exit_hook;    /*** long_jump support      ***/
    void             (*funct)();   /*** activity body          ***/
    mp_mem_p         param;        /*** activity parameters    ***/

    int         size;              /***
                                    *** size of parameter block freed
                                    *** upon joining.
                                    ***/

    int         join_cntr;         /***
                                    *** number of children this
                                    *** activity will join.
                                    ***/

    struct acb  *join_perf;        /***
                                    *** the activity who joins this
                                    *** activity
                                    ***/

    int         act_id;            /***
                                    *** unique integer identifying
                                    *** this activity.
                                    ***/
#ifdef  DEBUG
    int         aid;
    mp_name          name;         /*** activity name ***/
    mp_pointer       wblk;         /***
                                    *** if waiting or idle,
                                    ***  this shows where
                                    ***/

#endif  DEBUG
}   mp_acb, *mp_acb_p;

#define NOACB (mp_acb_p) 0
```

Figure 4-6: The activity control block structure

```
typedef struct {
    mp_lock         lock;         /***
                                   *** lock used when global state,
                                   *** ie, the workload control
                                   *** block, is updated.
                                   ***/

    mp_cond         need_proc;    /***
                                   *** condition used when processes
                                   *** are waiting to run activities
                                   ***/

    mp_queue        act_queue;    /***
                                   *** queue of activities waiting
                                   *** to be run.
                                   ***/

    int             pcount;       /*** process count          ***/
    int             acount;       /*** activity count         ***/
    int             doomsday;     /***
                                   *** set up for exiting from
                                   *** main
                                   ***/

    mp_lock         mem_lock;     /*** memory manager lock    ***/
    memun           buckets[32];  /*** memory manager buckets ***/
    unsigned short  act_id_cnt;   /*** activity id counter    ***/

#ifdef  MONITOR[4]
    unsigned int    mp_km_size;   /***
                                   *** Used when calling the
                                   *** kernel monitor to set the
                                   *** size of the kernel buffers.
                                   ***/
#endif  MONITOR

#ifdef  DEBUG
    int             tracecount;   /***
                                   *** counter used to order traced
                                   *** events
                                   ***/

    mp_name         name;         /*** workload name          ***/
    mp_cond         rip;          /*** dead acbs              ***/
    mp_lock         debug_lock;   /*** lock for breakpoint    ***/
    mp_queue        *trace_list;  /*** trace command queue    ***/
    mp_queue        *break_list;  /*** break command queue    ***/
    mp_acb_p        tree_root;    /*** head of dynamic tree   ***/
    mp_lock         tree_lock;    /*** dynamic tree lock      ***/
    long            last_event;   /*** last event traced      ***/
    mp_pointer  where;            /*** where am i             ***/
#endif  DEBUG
}               mp_wcb, *mp_wcb_p;
```

Figure 4-7: The workload control block structure

[4] May not be supported on all systems.

4.1.2. Standard MPC run-time functions

In addition to the standard set of run-time structures, there are several standard run-time functions that execute the parallelism of MPC. Below is a discussion of these standard run-time functions.

4.1.2.1. Workload organization: horses and riders

The MPC run-time assumes that in steady state there is a set of ready processes waiting for the condition `need_process` to appear. When this condition is signaled, the first of the ready processes will be assigned to an *acb* from the activity queue, `mp_wcb.act_que`, and start to run the activity. When the activity exits, the corresponding process will be released and will pick up another activity to run, or will wait depending on the state of the activity_que.

Think of activities as horse-back riders sitting on a corral fence and processes are horses wandering in and out of the corral looking for riders to carry. If a horse enters the corral and a rider is sitting on the fence (ie the `need_process` condition is set), the rider jumps on the horse and off they go somewhere out there (The activity grabs the process and begins executing). If there are no riders sitting around, the horse just hangs out in the corral until a rider appears (An activity is spawned). When a rider and horse come back to the corral, the rider gets off and goes away (When the activities finish, the activities drop the processes). If there is another rider waiting on the fence, he grabs the horse and off they go (If another activity exists, it grabs the freed process). After all the riders return, the horses are rounded up and everybody goes home (After all the activities exit, all the processes terminate).

4.1.2.2. Queue related functions

There are a variety of queue related functions: `mp_queue_init`, `mp_push_front`, `mp_push`, `mp_pop`, `mp_peek`. Each of the functions takes a parameter of type, `mp_queue_p` which is pointer to a queue. Figure 4-8 shows the queuing functions of MPC. Two of the functions, `mp_push_front` and `mp_push`, take an additional parameter `mp_item_p`, a pointer to the object to be place in the queue.

A queue is initialized to be empty by the `mp_queue_init` function. `mp_push_front()` adds the item to the front of the queue. `mp_push()` adds the item to the end of the queue. `mp_pop()` retrieves item on the front of the queue. `mp_peek()` returns what is on the front of the queue, but does not alter the queue's contents.

4.1.2.3. Lock related functions

There are four lock related functions: `mp_lock_init`, `mp_close`, `mp_open`, `mp_test_and_lock()`. Each of the functions take a parameter of type `mp_lock_p` which is a pointer to an `mp_lock`. Figure 4-9 shows the lock related functions of MPC.

`mp_lock_init()` initializes a lock to be open. `mp_close()` attempts to grab the lock passed to it; if it can't, it blocks the activity that called it until the lock is released. The activity is blocked by busy waiting for a pre-specified duration, and then is switched out. `mp_open()` releases the lock passed to it. `mp_test_and_lock()` will grab the lock passed to it if it is open, but returns control to the calling activity

```
void
mp_queue_init(q{,name})
mp_queue_p q;
mp_name name;

void
mp_push_front(q,i);
mp_queue_p q;
mp_item_p i;

void
mp_push(q,i);
mp_queue_p q;
mp_item_p i;

mp_item_p
mp_pop(q);
mp_queue_p q;

mp_item_p
mp_peek(q);
mp_queue_p q;
```

Figure 4-8: Queuing functions

```
void
mp_lock_init(lock{,name})
mp_lock_p lock;
mp_name name;

void
mp_close(lock)
mp_lock_p lock;

void
mp_open(lock)
mp_lock_p lock;

int
mp_test_and_lock(l)
mp_lock_p   *l;
```

Figure 4-9: Locking related functions

if the lock is already held. It returns the lock value regardless of whether the lock was grabbed or not. (ie. 0 if the lock is grabbed and 1 if it is already held). The parameters in braces are applicable if the DEBUG option is invoked.

4.1.2.4. Condition related functions

There are five condition related functions: `mp_cond_init()`, `mp_wait()`, `mp_mon_wait()`, `mp_signal_first()` and `mp_signal_all()`. All of the functions take a parameter of type, `mp_cond_p` which is a pointer to a condition. In addition, the functions, `mp_wait()` and `mp_mon_wait()` take another parameter of type, `mp_lock_p` which is a pointer to a lock. Figure 4-10 shows the condition related functions of MPC.

```
void
mp_cond_init(c{,name})
mp_cond_p c;
mp_name name;

void
mp_wait(c,l)
mp_cond_p c;
mp_lock_p l;

void
mp_mon_wait(c,l)
mp_cond_p c;
mp_lock_p l;

mp_item_p
mp_signal_first(c)
mp_cond_p c;

void
mp_signal_all(c)
mp_cond_p c;
```

Figure 4-10: Condition related functions

mp_cond_init() initializes the lock and queue of a condition. mp_wait() and mp_mon_wait() each put a pointer to the calling activity's *acb*[5], mp_acb_p, in the queue of the condition passed to it. They differ only in how they behave when the monitor library is used in that mp_wait contains special sensors while mp_mon_wait does not. Both use a lock to ensure atomic action in the following manner:

1. Although it is not necessary in all case, the caller usually grabs the lock that it intends to pass before calling these either functions. The caller passes the pointer to the lock.
2. The lock is released after the pointer is pushed into the conditions queue and the activity is rescheduled as blocked.
3. The lock is closed again after the activity is unblocked in order for housekeeping to be done in an atomic way.
4. Upon return from these calls, the calling activity is allowed to continue execution. The caller is expected to *release* the lock after these wait calls return.

mp_signal_first() sends out a condition signal to first activity in the condition queue of the condition passed to the function. Any other activities in the queue are not signalled and must wait further. mp_signal_all() sends the signal to all the activities waiting in the condition queue. All of the activities are then allowed to continue processing. Parmeters in braces apply to the DEBUG option.

[5]See the discussion of Activity Control Blocks above

4.1.2.5. Synchronization functions

There are five synchronization functions: mp_opr_init, mp_sync, mp_release, mp_dsync, mp_drelease. Figure 4-11 shows the synchronization related functions of MPC.

```
void
mp_opr_init(c{,name});
mp_opr_p        c;
mp_name       name;

void
mp_sync(lock,a,paramlist)
mp_lock_p        lock;
mp_opr_p         a;
int      paramlist;

void
mp_dsync (lock, paramlist)
mp_lock_p        lock;
mp_lock_p        paramlist;

void
mp_release(lock,a,paramlist)
mp_lock_p        lock;
mp_opr_p         a;
int      paramlist;

void
mp_drelease(lock, paramlist)
mp_lock_p        lock;
mp_lock_p        paramlist;
```

Figure 4-11: Synchronization related functions

mp_opr_init() performs initialization functions for operations. It takes a pointer to a mp_opr variable as a parameter which represents the operation in the frame data structure. mp_sync() is used to implement MPC sync calls. It takes a pointer to a frame lock, followed by the pointer to the corresponding synchronizing operation and pointers to every operation being synchronized upon. The caller blocks if the corresponding synchronized statements are currently being executed. The caller will unblock when all the current execution of the synchronized statements finishes. The parameters required by mp_release() are the same as those taken by mp_sync(). The function is used at the end of operations that synchronize on other operations to inform the other operations that the calling operation has finished. mp_dsync() and mp_drelease() are used to implement dsync call. They take a list of locks to be grabbed and released atomically. The lock supplied as the first parameter is a general lock to ensure the atomicity of the function.

4.1.2.6. Global heap related functions

MPC run-time keeps its own global heap. There are four functions for managing global memory: mp_alloc, mp_free, mp_calloc and mp_realloc. Figure 4-12 shows the memory management functions of MPC.

```
memory_p
mp_alloc(size)
int size;

memory_p
mp_calloc(size)
int size;

memory_p
mp_realloc(ptr,size)
memory_p ptr;
int size;

void
mp_free(base)
char *base
```

Figure 4-12: Memory management functions

Memory blocks can only be allocated and deallocated from the global heap by using `mp_alloc()` and `mp_free()`[6]. `mp_alloc()` takes an integer parameter, `size`, which designates the number of bytes to requested. It returns a pointer to a byte addressable chunk of global memory. Because memory management information is stored with each allocation, the allocated size is at least `sizeof(mp_mem)` bytes. Although this information is included, the pointer that is returned points only to the usable part of the allocated space. Since `mp_alloc` allocates memory in chunks of powers of 2, the size received is sometimes considerably larger than that requested.

`mp_free()` takes a pointer to the start of the memory block which is to be freed. A new cleared memory block could be obtained by using `mp_calloc` call, and any already allocated global memory block can be reallocated and resized by `mp_realloc()` function.

In addition to the memory space global to the whole parallel workload, each of the parallel activities keeps its own local space which is visible to the all functions called from within the activity. This space can be allocated via standard memory allocation calls (ie. `malloc()`, `realloc()`, etc.).

4.1.2.7. Activity and workload related functions

Control of activities and operations is done by calling workload related functions. These are: `mp_workload_init`, `mp_activity`, `mp_join`, `mp_detach`, `mp_entry` and `mp_exit`. Figure 4-13 shows the activity and workload related functions of MPC.

The function `mp_workload_init()` is the first MPC function executed in any program linked with any of the three MPC run-time libraries. It initializes the workload structures. This call requires three parameters:

- g - size of the global heap in mega-bytes.

---

[6]Using standard `malloc()` calls will only allocate memory local to the activity calling `malloc()`.

```
void
mp_workload_init(g,y,p);
int g,y,p;

mp_acb_p
mp_activity({name,}funct,size,paramlist)
mp_name         name;
void (*funct)();
int size;
char paramlist;

void
mp_join(act_list)
mp_acb_p act_list;

void
mp_detach(act)
mp_acb_p act;

void
mp_entry(ptr,paramlist)
char *ptr;
int paramlist;

void
mp_exit(paramlist)
int paramlist;
```

Figure 4-13: Activity and workload related functions

- y - number of spins to be taken by locks before yielding
- p - number of processes to be created at the run-time.

`mp_activity()` creates and spawns activities. It is called with a pointer to the function, `(*funct)()`, which comprises the activity, the size, `size`, in bytes of the activity's data structure, and a list, `paramlist`, of information about the variables in the activity. For each variable, this list contains a pointer to the variable and the size of the variable. The list is terminated with a NULL. If the DEBUG option is selected, the function is passed the name, `name`, of the activity to be made. `mp_join()` takes as parameters a list,`act_list`, of pointers to the activities to be joined which it joins. The calling activity is blocked until all of the activities in the list have completed execution. `mp_detach()` takes the same parameters as `mp_join()`. The activities is creates and spawns can never be joined. `mp_entry()` is called when an activity is entered. It takes as parameters a pointer to the activity, and a list of all the variables in the activity's variable declaration list. The list constains a pointer to each variable followed by the size of the variable. `mp_exit()` is called upon exit of an activity and takes as a parameter exit condition codes. This is most commonly NULL.

4.2. Some programming tricks

All MPC run-time structures and functions are accessible by a programmer. Since MPC syntax is implemented using these run-time objects, it is possible for the programmer to program by using these objects directly. Doing so brings only modest, if any, performance improvement and increases the complexity of a programmer's work. Consequently, in many cases it is not recommended that the programmer attempt to circumvent MPC syntax by programming in this manner.

4.2.1. Activity identification: using my_act

There, however, some structures and functions that a more adventurous programmer may want to use. For example, as discussed in section 4.1.1.8 a programmer can access his activity control block via `my_act`. One of the useful members of this structure is `act_id` which is an integer, unique to each activity. This variable can be useful in implementing objects like parallel buffers, in cases when it is desirable that buffer contention be reduced. Such an implementation would consist of an array of buffers, perhaps one per activity, with `my_act->act_id` as the index into the array:

```
buffer[my_act->act_id] = item;
```

where `item` is some object to be placed into the buffer.

The `act_id` of each activity is assigned using the value of a counter in the workload control block, `workload->act_id_cnt` when the activity is created. This counter may be changed in the following manner:

```
mp_close(&workload->lock);
workload->act_id_cnt = (int) x;
mp_open(&workload->lock);
```

where `x` has an arbitrary (non-negative) value. The activity changing the counter value must grab the workload lock to prevent races.

4.2.2. Using locks and conditions

The MPC `sync` and `dsync` functions are powerful synchronization primitives. There may be times, however, when either these functions simply do not provide the functionality the programmer desires or he does not know how use these functions to implement some protocal he has in mind. An example of the first case is when a kind of test-and-set primitive is needed. For the second case, perhaps he merely wishes to have a clear way to send signals between activities.

4.2.2.1. A test-and-set function

The MPC run-time function, `mp_test_and_lock()`, performs the atomic test-and-set operation. Figure 4-14 shows an example of using this function.

Here, `user_lock` is of the type `mp_lock`. If `user_lock` is to be shared between activities, *it must be declared and accessed as a global variable in a frame!* If the lock is not declared as a global frame variable, different activities may reserve different memory for the lock. In frame shown in Figure 4-14, if

```
frame
buffer()
{
   mp_lock       user_lock;

   opr
   calculate()
   {
        if (mp_test_and_lock(&user_lock)
        {
            ----------
            ---------
            ------------
            mp_close(&user_lock);
        }
        -------
        ----
   }

   {
        ------
        ---
        mp_lock_init(&user_lock);
   }
}
```

Figure 4-14: An Example of using mp_test_and_lock

user_lock is already held, mp_test_and_lock returns a 1; if not, it grabs user_lock and returns a 0. Here, after the lock is checked, the caller continues if the lock is not held. In this case, when the lock is already held, the caller executes some arbitrary statements before calling mp_close() in order to block on the lock. Notice that user_lock is initialized in the initialization section of the frame.

4.2.2.2. Signalling

Figure 4-15 shows a case where both a lock and a condition are used. Here, the writer waits for a signal from the reader indicating that the buffer in question is not full (presumably, the writer would have performed some tests to indicate that the buffer is full before it executes the mp_wait). As described in section 4.1.2.4, the lock waiting_for_reader is usually grabbed before the wait is called (this is not necessary if atomicity is not important). The call to mp_open, however, is necessary. Notice that the lock and condition are initialized in the initialization section of the frame.

4.2.2.3. Dynamic memory allocation

The global memory management functions are discussed in section 4.1.2.6. It is important to keep in mind how these these functions differ from the standard C memory allocation functions. Global memory blocks can only be allocated and deallocated using mp_alloc() and mp_free(). In addition to this global memory space, each of the parallel activities keeps its own local space which is visible to the all functions called from within the activity. This space can be allocated via standard memory allocation calls (ie. malloc(), realloc(), etc.). Using the standard malloc() call, for example, only allocates memory local to the activity calling malloc(). That is, if an activity calls malloc() the memory allocation it receives can be correctly referenced only by statements or functions within that activity.

```
frame
buffer()
{
   mp_lock        waiting_for_reader;
   mp_cond        buffer_is_not_full;
   --------       ---------
   ---            -----

   opr
   write()
   {
        ---------
        mp_close(&waiting_for_reader);
        -----
        mp_wait(&buffer_is_not_full, &waiting_for_reader);
        mp_open(&waiting_for_reader);
        -------
        -----
        ----
        --------
   }

   opr
   read()
   {
        --------
        ---
        --------
        mp_signal_first(&buffer_is_not_full);
        ------
   }
   {
        -----


        ------
        mp_lock_init(&waiting_for_reader);
        mp_cond_init(&buffer_is_not_full);
        ----
   }
}
```

Figure 4-15: Example of using locks and conditions

4.3. Skeleton of an MPC generated C file.

The C file generated by the MPC preprocessor from a MPC program contains several data structures and function calls not present in the MPC file. In Figure 4-16 is the skeleton of a MPC program. It consists of an instantiated frame, `matrix()` with a single operation, `get()`, and one global variable, `(float) z`. Note that the operation exports the global variable. The program also has a single activity, `multiply()`, which executes the frame operation once. Finally, in `main()`, the activity is instantiated and spawned once. Let's call this program `mat.mpc`; a MPC command line that could generate the following C file is:

```
mpc mat.mpc -o mat --m --C -lm
```

The —m switch means that monitoring is enabled and the —c switch means that a C file will be printed in the immediate directory. The -lm switch might not be necessary, but since the monitor automatically includes the block timers, the user may have inserted some timers in his code.

In Figure 4-17 is the top half of a sparse skeleton of the corresponding C file. Much of the C file consists of data declarations (as much as 80% or 90% sometimes). Many of these declarations are present as the result of MPC expanding several include files into the C file. In this case, MPC inserts a special include file at the top of the C file and then dumps the data declarations, not all of which are shown.

After the data declarations, come various frame initialization functions. Such a function is generated by the MPC preprocessor in order to initialize a particular frame. In this case, the program uses three frames: one for the block timers, one for the performance monitor, and the mult frame found in mat.mpc. These functions are called later at the start of main. Deeper into the C file, is a structure declaration, mp_multiply_para_type which represents the parameters of the multiply activity. This is followed by the functional definition of the activity mp_multiply() shown in Figure 4-18. In this function is a call to mp_entry which copies the data in mp_multiply_para into the local variables.

Since this is a monitored program (presumably the user ran mat.mpc through PIEmacs), the C file contains sensor macros. The activity is delimited by Activity_begin() and Activity_end sensors. The antecedents of conditional statements that contain them test a sensor enable table to determine if the sensors are enabled. The constants being passed to the sensor are for identification purpose later when an execution is viewed by PIEscope. There are also frame operation sensors delimiting the frame operation. The implementation of the export() statement is shown inside this frame. As may be noticed, each of the sensor macros include a "1" concatenated to the names of the sensors. This is an implementation detail that is of no concern to a programmer.

In main() can be seen the calls to workload_init()[7] and the frame initialization functions. There are also calls to mp_set_exit(), sensinit(), slurp_runtime_enable_table() and mp_enab_set(). mp_set_exit() sets up some exit conditions for main() while sensinit() initializes some monitoring structures and spawns the collector. slurp_runtime_enable_table() reads SEP file[8] and stores its sensor enabling information in the sensor_enable_table. mp_enab_set() enables the run-time sensors. After these setup calls, the body of main is executed.

---

[7]See section 4.1.2.7

[8]The SEP file is the file containing sensor enable information.

```
frame
matrix(m, n)
int              m, n;
{
    float            z;

    opr float  get(x, y)
    int            x, y;
    {
        ------
        ----
        export(z);
    }
    {}
} mult(SIZE, SIZE);

act
multiply(x1, x2, y1, y2, mx, my, sz)
int      x1, x2, y1, y2, mx, my, sz;
{
    --------
    x = mult.get(k,i);
    -------
      ------
    --------
};

main(argc, argv)
int             argc;
char            **argv;
{
    multiply    task;
    int         s, x, y;

    --------
    ------------
    task(0, s - 1, 0, s - 1, x, y, s);
    -----
      --------
    --------
}
```

Figure 4-16: Skeleton of an MPC program

```
#include <mpc_def.h>
typedef int      jmp_buf[10];
typedef char    *mp_name;
                  .
                  .
                  .
void
mp_mp_block_time_init(frame, id)
        mp_mp_block_time_e_type *frame;
        int                 id;
{
    -------
      -------
    ------
} mp_mp_block_time_type *mp_mp_block_time;
                  .
                  .
                  .
void
mp_mp_monitor_init(frame, id)
    mp_mp_monitor_e_type *frame;
    int                 id;
{
    ---------
    -------
      ---------
} mp_mp_monitor_type *mp_mp_monitor;
                  .
                  .
                  .
void
mp_mult_init(frame, id)
        mp_mult_e_type    *frame;
        int                id;
{
    ---------
    -------
      ---------
} mp_mult_type *mp_mult;
                  .
                  .
                  .
typedef struct
{
    int  x1, x2, y1, y2, mx, my, sz;
    int  mp_ret_val;
} mp_multiply_para_type;
```

Figure 4-17: Skeleton of first half of resulting C file

```
int
mp_multiply(mp_multiply_para)
mp_multiply_para_type *mp_multiply_para;
{
    int  x1 ,  x2 ,  y1 ,  y2 ,  mx ,  my ,  sz ;
    mp_entry(mp_multiply_para,&(x1), sizeof(x1)  ,&(x2), sizeof(x2),
                              &(y1), sizeof(y1)  ,&(y2), sizeof(y2),
                              &(mx), sizeof(mx)  ,&(my), sizeof(my) ,
                              &(sz), sizeof(sz)   , NIL);
    if (sensor_enable_table?sensor_enable_table[47]?
        sensor_enable_table[47][5]:0:0) mp_Activity_begin1(5, 47);
    {
        --------
        if (sensor_enable_table?sensor_enable_table[47]?
            sensor_enable_table[47][2]:0:0) mp_Fop_begin1(2, 47);
        {
            ---------
            x = ((float) (* mp_mult).z);
            --------
            ------
        }
        if (sensor_enable_table?sensor_enable_table[47]?
            sensor_enable_table[47][2]:0:0) mp_Fop_end1(2, 47);
        -----------
        --------
    }
    if (sensor_enable_table?sensor_enable_table[47]?
        sensor_enable_table[47][5]:0:0) mp_Activity_end1(5, 47);
    mp_exit(NIL);
}

main(argc, argv)
int     argc;
char    **argv;
{
   mp_workload_init(2, 200, 16);
   mp_set_exit();
   mp_mp_block_time = (mp_mp_block_time_type *)
     mp_init_frame(sizeof(mp_mp_block_time_e_type),
                   0, mp_mp_block_time_init);
   mp_mp_monitor = (mp_mp_monitor_type *)
     mp_init_frame(sizeof(mp_mp_monitor_e_type),
                   0, mp_mp_monitor_init);
   mp_mult = (mp_mult_type *)
     mp_init_frame(sizeof(mp_mult_e_type), 1, 3, mp_mult_init);
   sensinit(44);
   sensor_enable_table = slurp_runtime_enable_table("SEP");
   mp_enab_set(1);
   ------------
   task = mp_activity(mp_multiply ,sizeof(mp_multiply_para_type),
                      0, s - 1, 0, s - 1, x, y,s);
   -----------
     -----
   mp_exit(NIL);
}
```

Figure 4-18: Skeleton of second half of resulting C file

Appendix I
Count.mpc

Count is a very simple example that is called with

```
Count
```

The program creates two activities that each increment a global variable. The operation `inc` is used to do this, and as the program is listed here, it contains a sync statement that will cause the process that calls `inc` second to wait until the first process to finish incrementing the variable. As a result one process will increment the counter 10 times, exit, and allow the other process to increment the counter 10 more times. If the sync statement is removed from the operation, the processes will run in parallel, and the incrementing of the counter will be interleaved between the two processes until each has incremented the counter ten times.

```
/***
        This is to test sync statement
***/

frame global(k)
int k;
{
        int i;
        opr inc(x)
        int x;
        {
                int j,z;
/***
        in order to see the difference sync makes the
        test should be run twice: once with sync
        commented out
***/
                sync(inc){

                    for(z=0;z<10;z++)  {
                        printf("act %d: %d\n",x,i++);
                        sleep(1);
                    }
                }
                export();
        }
        {
                i = k;
        }
} glob(10);
act incr(x)
int     x;
{
        glob.inc(x);
} e[3];
```

```
main()
{
        incr f[2];
/***
        Main will start two activities. Each will
        call frame operation giving its id as the
        input parameter. In the case with sync
        printouts on the screen should be ordered
***/
        printf("Program start...\n");
        printf("First activity call...\n");
        e[0](1);
        printf("Second activity call...\n");
        f[1](2);
        join(e[0],f[1]);
        printf("End of main...\n");
}
```

Appendix II
Varcount.mpc

Varcount is like the Count example, except that the user can vary the number of activities which are running and can vary whether the activities should run *synchronously* or *asynchronously.*

Varcount is called with

```
varcount
```

The program initially creates two activities running *synchronously.* Once it is running, the user can type to the keyboard any number from 1-16 (inclusive) to indicate the number of parallel activities to run on the next pass. Typing 17 causes the activities to run asynchronously and typing 18 causes the activities to run synchronously again. Input of -1 causes the program to exit.

```
/***
  This is to test sync statement.
  ***/
#include <stdio.h>

frame global(k)
     int k;
{
  int count, nact, ctrlflag;

  opr void inc(x)
    int x;
  {
    int j,z;
    /***
      in order to see the difference sync makes the
      test should be run twice: once with sync
      commented out
      ***/
    if (ctrlflag == 1) {
      sync(inc)
        {
          for(z=0;z < 10;z++)
            {
              printf("act:%d, count=%d\n",x,count++);
              sleep(1);
            }
        }
    } else {
      for(z=0;z < 10;z++)
        {
          printf("act:%d, count=%d\n",x,count++);
          sleep(1);
        }
    }
  }
```

```
    opr int control()
      {
        export(ctrlflag);
      }

    opr int numact()
      {
        export(nact);
      }

    opr init(val)
      int val;
    {
      export(count = val);
    }

    {
      count = k; nact = 2; ctrlflag = 1;
    }

  } GLOB(0);



  act incr(x)
       int         x;
  {
    GLOB.inc(x);
  } counter[16];
```

```
act interface()
{
  int num;

  while(1) {

    scanf("%d",&num);
    if (num == 17) {
      GLOB.control() = 0;
      continue;
    }
    if (num == 18) {
      GLOB.control() = 1;
      continue;
    }
    if (num == 19) {
      GLOB.control() = 2;
      continue;
    }
    GLOB.numact() = num;
    if (num <= 0 || num > 16) exit();
  }
}interf;
```

```
main()
{

  int limit, i;
  /***
    Main will start 1 - 16 activities depending on.  Each of them
    will call frame operation increment 10 times and exit;
    ***/

  printf("Program start...\n");
  printf("Start interface: Enter number of activities any time\n");
  printf("Valid numbers of activities are 1 - 16\n");
  printf("Typing 17 causes activities to run asyncronously\n");
  printf("Typing 18 causes activities to run syncronously\n");
  printf("Typing -1 causes the program to exit.\n");

  GLOB.numact() = 2;
  interf();

  while (1) {
    i = GLOB.control();
    while (i == 2) {
      i = GLOB.control();
    }
    printf("Start of current pass...\n");
    GLOB.init(0);
    limit = GLOB.numact();
    if (limit <= 0 || limit > 16) {
      printf("Out of limit ... will exit\n");
      break;
    }
    for (i = 0; i < limit;i++)
      counter[i](i);
    for (i = 0; i < limit;i++)
      join(counter[i]);
    printf("End of current pass...\n");
  }
  join(interf);
  printf("End of workload...\n");
}
```

Appendix III
Matrix.mpc

This example program will take two user-entered square arrays and multiply them together. It is called with a command like:

```
matrix size dx dy
```

where size is the length of the sides of the arrays (the number of elements in each array is size * size). Dx and dy are the minimum desired sizes (size is as defined before) for the submatrices that the program will create from the larger matrices. For instance, if the command

```
matrix 12 4 4
```

is entered, the program will expect the user to input 144 elements for each of the two arrays to be multiplied. The elements must be integers separated by spaces and each row must be terminated with a carriage return. The first array must be followed by a '@' and another blank line before the second array can be entered. The entered arrays will be cut into fourths (each side will be cut in half) and submit the portions to subtasks which will continue dividing until the length of a vertical side of a portion is smaller than dy (in this case 4) and the horizontal side is smaller than dx (4 again). In this example 3 cuts must be made to each side, and therefore 16 subtasks will be created for each of the sixteen portions of the divided array.

When the array has been divided up, each subtask multiplies its own parts of the two arrays and adds the result to the proper place in the result array. After all the subtasks have finished with their multiplication the result array contains the result of the multiplication of the two user-entered arrays.

```
#include <libc.h>

/*******************************************************

  This is the matrix multiplication example to
  to test the syntax of frames and activities

*******************************************************/

#define SIZE 128
#define NUM  2


/***
    Frame matrix is the matrix template. It exports two operations,
    'get' and 'put'.
 ***/

/***
    Operations are self explanatory. The question could be raised:
    "Why is there no sync defined for put?"  In the general case it
    should obviously be "sync(put){}". The reason it is omitted here
    lies in the fact that we know that only one activity is going to
    write any one particular matrix element, so sync would just add
    unnecessary overhead.
***/
```

```
frame
matrix(m, n)
     int              m, n;
{
  float             matrix_data[m][n];

  opr float         get(x, y)
    int               x, y;
  {
    export(matrix_data[x][y]);
  }

  opr float         put(x, y)
    int               x, y;
  {
    export(matrix_data[x][y]);
  }

  {
  }
} a(SIZE, SIZE)[3];
/***
    a(...)[3] is a shorthand to create three instances with the same
    initial parameters.  Another way of doing it would be:

        matrix a1(SIZE,SIZE), a2(SIZE,SIZE), a3(SIZE,SIZE);
***/
```

```
act
multiply(x1, x2, y1, y2, mx, my, sz)
/***
    Gets the submatrix described by x1,x2,y1, and y2 and checks
    its dimensions against given limits mx and my. If any submatrix
    dimension is larger than mx or my respectively will cut the
    submatrix along this dimension in two halves, starts two new
    multiply activities (ie. subtasks), and gives them submatrices to
    work on. When the both limits are satisfied(ie. there is no more
    need to cut), multiply will do the multiplication and join the
    father activity upon completion.
***/
        int             x1, x2, y1, y2, mx, my, sz;
{
  int               ex, ey, i, j, k;
  float             t, tmp, tmp2;
  multiply          subtask[2];

  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;
  if (ex > ey) {
    /***
      x dimension of the submatrix is larger one
      ***/
    if (ex > mx) {
      /***
        x dimension of the submatrix is larger than mx limit
        (means we have to cut it in two halves)
        ***/
      subtask[0] (x1, (x1 + ex / 2) - 1, y1, y2, mx, my, sz);
      subtask[1] ((x1 + ex / 2), x2, y1, y2, mx, my, sz);
      join(subtask[0], subtask[1]);
      exit();
    }
    if (ey > my) {
      /***
        y dimension is larger than my limit
        ***/
      subtask[0] (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
      subtask[1] (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
      join(subtask[0], subtask[1]);
      exit();
    }
  }
```

```
  if (ey >= ex) {
    if (ey > my) {
      subtask[0] (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
      subtask[1] (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
      join(subtask[0], subtask[1]);
      exit();
    }
    if (ex > mx) {
      subtask[0] (x1, (x1 + ex / 2 - 1), y1, y2, mx, my, sz);
      subtask[1] ((x1 + ex / 2), x2, y1, y2, mx, my, sz);
      join(subtask[0], subtask[1]);
      exit();
    }
  }
  /***
    NO more subdivisions! DO THE JOB!!
    ***/
  for (i = y1; i <= y2; i++) {
    for (j = x1; j <= x2; j++) {
      t = 0;
      for (k = 0; k < sz; k++) {
        tmp  = a[0].get(i,k);
        tmp2 = a[1].get(k,j);
        t = t + tmp * tmp2;
      }
      a[2].put(j, i) = t;
    }
  }
  /***
    The product submatrix calculated
    ***/
};
```

```
init_matrices(sz)
        int             sz;
{
  int               x, y;
  float             tmp;

  printf("enter matrices row by row\n");
  printf("row separator is CR, and matrix separator is @CR\n");
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      scanf("%f", &tmp);
      a[0].put(x, y) = tmp;
    }
    while (getchar() != '\n');
  }
  while (getchar() != '@');
  while (getchar() != '\n');
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      scanf("%f", &tmp);
      a[1].put(x, y) = tmp;
      a[2].put(x, y) = 0;
    }
    while (getchar() != '\n');
  }
}



print_result(sz)
        int             sz;
{
  int               x, y;
  float             tmp;

  printf("\nresult:\n");
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      tmp = a[2].get(x, y);
      printf("%0.1f ", tmp);
    }
    printf("\n");
  }
}
```

```
main(argc, argv)
        int              argc;
        char           **argv;
{
  int               sz, mx, my;
  multiply          task;

  if (argc != 4) {
    fprintf(stderr, "Usage: matrix size dx dy\n");
    exit();
  }
  sz = atoi(argv[1]);
  mx = atoi(argv[2]);
  my = atoi(argv[3]);
  init_matrices(sz);
  task(0, sz - 1, 0, sz - 1, mx, my, sz);
  join(task);
  print_result(sz);
}
```

60

Appendix IV
Newmat.mpc

This example is just like the matrix example, except in the way it divides the work. In the matrix example, whenever an activity decided it should sub-divide itself, it would create two children, give each half of the work, and then wait for them to finish. This meant that the parent activity was not doing work, and is using resources by its very existence.

In this example, the parent creates only one child, and keeps half of the work for himself by calling a recursive procedure. We found that this algorithm runs about 20% faster than the example shown in the matrix example.

Newmat will take two user-entered square arrays and multiply them together. It is called with a command just like in the matrix example:

```
newmat size dx dy
```

where the parameters have the same meaning as in the matrix example.

```
#include <libc.h>

/******************************************************

  This is the matrix multiplication example to
  to test the syntax of frames and activities

******************************************************/

#define SIZE 160
#define NUM  2

/*
 * Frame matrix is the matrix template. It exports two operations,
 * "get" and "put".
 */

/*
 * Operations are self explanatory. The question could be raised:
 * "Why is there no sync defined for put". In the general case it
 * should obviously be "sync(put) {}". The reason it is omited here
 * lies in the fact that we know that only one activity is going to
 * write particular matrix element, so sync would just add some
 * unnecessary overhead.
 */
frame
matrix(m, n)
     int        m, n;
{
  float             matrix_data[m][n];

  opr float         get(x, y)
    int               x, y;
  {
    export(matrix_data[x][y]);
  }
  opr float          put(x, y)
    int               x, y;
  {
    export(matrix_data[x][y]);
  }

  {
  }
} a(SIZE, SIZE)[3];
/*
 * a(....)[3] is shorthand to create three instances with the same
 * initial parameters. Another way of doing it would be: matrix
 * a1(SIZE,SIZE),a2(SIZE,SIZE),a3(SIZE,SIZE);
 */
```

```
act
multiply(x1, x2, y1, y2, mx, my, sz)
     int         x1, x2, y1, y2, mx, my, sz;
{
  multproc(x1,x2,y1,y2,mx,my,sz);
};



multproc(x1, x2, y1, y2, mx, my, sz)
/***
        Gets the submatrix described by x1,x2,y1, and y2 and
checks its dimensions against given limits mx and my. If any
submatrix dimension is larger than mx or my respectively will
cut the submatrix along this dimension in two halves, starts two
new multiply activities (ie. subtasks), and gives them submatrixes
to work on. When the both limits are satisfied(ie. there is no more
need to cut), multiply will do the multiplication and join the father
activity upon completion.
***/
     int              x1, x2, y1, y2, mx, my, sz;
{
  int              ex, ey, i, j, k;
  float            t, tmp, tmp2;
  multiply         subtask;

  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;
  if (ex > ey) {
    /***
      x dimension of the submatrix is larger one
      ***/
    if (ex > mx) {
      /***
        x dimension of the submatrix is larger than mx limit
        (means we have to cut it in two halves)
        ***/
      subtask (x1, (x1 + ex / 2 - 1), y1, y2, mx, my, sz);
      multproc((x1 + ex / 2), x2, y1, y2, mx, my, sz);
      join(subtask);
      return;
    }
    if (ey > my) {
      /***
        y dimension is larger than my limit
        ***/
      subtask (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
      multproc (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
      join(subtask);
      return;
    }
  }
```

```
    if (ey >= ex) {
      if (ey > my) {
        subtask (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
        multproc (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
        join(subtask);
        return;
      }
      if (ex > mx) {
        subtask (x1, (x1 + ex / 2 - 1), y1, y2, mx, my, sz);
        multproc ((x1 + ex / 2), x2, y1, y2, mx, my, sz);
        join(subtask);
        return;
      }
    }
    /***
      NO more subdivisions! DO THE JOB!!
      ***/
    for (i = y1; i <= y2; i++) {
      for (j = x1; j <= x2; j++) {
        t = 0;
        for (k = 0; k < sz; k++) {
          tmp  = a[0].get(i,k);
          tmp2 = a[1].get(k,j);
          t = t + tmp * tmp2;
        }
        a[2].put(j, i) = t;
      }
    }
    /***
      The product submatrix calculated
      ***/
  }
```

```
init_matrices(sz)
    int             sz;
{
  int             x, y;
  float           tmp;

  printf("enter matrices row by row\n");
  printf("row separator is CR, and matrix separator is @CR\n");
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      scanf("%f", &tmp);
      a[0].put(x, y) = tmp;
    }
    while (getchar() != '\n');
  }
  while (getchar() != '@');
  while (getchar() != '\n');
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      scanf("%f", &tmp);
      a[1].put(x, y) = tmp;
      a[2].put(x, y) = 0;
    }
    while (getchar() != '\n');
  }
}



print_result(sz)
    int             sz;
{
  int             x, y;
  float           tmp;

  printf("\nresult:\n");
  for (x = 0; x < sz; x++) {
    for (y = 0; y < sz; y++) {
      tmp = a[2].get(x, y);
      printf("%0.1f ", tmp);
    }
    printf("\n");
  }
}
```

```
main(argc, argv)
     int              argc;
     char           **argv;
{
  int              sz, mx, my;
  multiply         task;

  if (argc != 4) {
    fprintf(stderr, "Usage: matrix size dx dy\n");
    exit();
  }
  sz = atoi(argv[1]);
  mx = atoi(argv[2]);
  my = atoi(argv[3]);
  init_matrices(sz);
  SENSOR("Before task");
  task(0, sz - 1, 0, sz - 1, mx, my, sz);
  join(task);
  print_result(sz);
}
```

Appendix V
Qsort.mpc

Qsort is called with the command:

```
qsort size
```

Size is the size of the array to be sorted, and after the line is entered, the program expects the user to enter Size number of integer elements separated by spaces or carriage returns. The array will then be broken down into parts that will be sorted according to the quick sort algorithm.

```
/*******************************************************************

        Example how to use frames and activities

        This is an implementation of the quick sort algorithm.

********************************************************************/

#include <stdio.h>

#define ARRAY_SIZE 100
#define TRUE 1



/*******************************************************************
        This is a template of a array data structure which
        include swap, put and get operation.
*******************************************************************/
frame array(n)
     int n;
{
  int array_data[n];
  /***
    operation 'swap' swaps two array elements.
   ***/

  opr void      swap(x, y)
    int x, y;
  {
    int              tmp;

    tmp = array_data[x];
    array_data[x] = array_data[y];
    array_data[y] = tmp;
    export();
  }

  opr int       put(i)
    int i;
  {
    export(array_data[i]);
  }
```

```
  opr int        get(i)
    int i;
  {
    export(array_data[i]);
  }

  {
  }
} qsarray(ARRAY_SIZE);



/*********************************************************************
        Activity sort implements quick sort algorithm.
*********************************************************************/
act sort(left, right)
     int left, right;
{
  int               j, k, tmp, tmp2;
  sort              subsort[2];

  if (left < right) {
    j = left;
    k = right + 1;
    do {
      do {
        j++;
        tmp  = qsarray.get(j);
        tmp2 = qsarray.get(left);
      } while ((tmp >= tmp2) && (j < right));

      do {
        k--;
        tmp  = qsarray.get(k);
        tmp2 = qsarray.get(left);
      } while ((tmp <= tmp2) && (k > left));

      if (j < k) {
        qsarray.swap(j, k);
      } else
        break;
    } while (TRUE);
```

```
    qsarray.swap(left, k);
    /******************************************************************
      Divide array and spawn two new sort activities.
      First part of the array (up to K-th element) has all
      elements greater then k-th element. Second part of the
      array (from k+1-th element) has all elements smaller then k-th
      element.
      ***************************************************************/
    subsort[0] (left, k - 1);
    subsort[1] (k + 1, right);
    join(subsort[0], subsort[1]);
  }
};



/****************************************************
        Initialize array and add sentinel
****************************************************/
init_array(n)
     int n;
{
  int               i, data, sum, tmp;

  sum = 0;
  for (i = 0; i < n; i++) {
    scanf("%d", &data);
    qsarray.put(i) = data;
    tmp = qsarray.get(i);
    sum = sum + tmp;
  }
  qsarray.put(n) = sum;
}



/****************************************************
        Print array of n elements
****************************************************/
print_array(n)
     int n;
{
  int               i, tmp;

  for (i = 0; i < n; i++) {
    tmp = qsarray.get(i);
    printf("%d ",tmp);
  }
  printf("\n");
}
```

```
main(argc, argv)
      int argc;
      int **argv;
{
  int             n, data;
  sort            qsort;

  if (argc != 2) {
    fprintf(stderr, "USAGE: qsort size\n");
    exit();
  }
  n = atoi(argv[1]);
  printf("Please enter %d integers\n", n);
  init_array(n);
  printf("\nARRAY:\n");
  print_array(n);
  qsort(0, --n);
  join(qsort);
  printf("\nRESULT:\n");
  print_array(++n);
}
```

Appendix VI
Sortm.mpc

Sortm is another parallel sort algorithm that is called with a command of the form:

```
sortm size subsize
```

As in Qsort, Size is the size of the array to be sorted and once the command line is entered, the program will expect the user to enter integer elements of the array separated by spaces or carriage returns. Subsize denotes the size of the subarrays that the program will divide the entered array into. The idea is that the program will take the array and cut it in half giving each half to subtasks. The subtasks will then halve the subarray if it is larger than the subsize entered by the user. When no more cutting is necessary, the subarrays are sorted with the merge sort algorithm. Then these sorted subarrays are sorted. This process is continued until the two original halves of the array are sorted by the original process with respect to each other and the array is completely sorted.

```
#include <libc.h>
/*******************************************************************

        This is an implementation of the sort-merge algorithm.

*********************************************************************/

#define ARRAY_SIZE 100
#define TRUE 1



/*******************************************************************
        This is a template of a array data structure which
        include swap, put, get and compare operations.
*******************************************************************/
frame array(n)
    int n;
{
  int               array_data[n];

  opr void          swap(i1, j1)
    int i1, j1;
  {
    int tmp, i11, j11;

    i11 = i1;
    j11 = j1;
    tmp = array_data[i11];
    array_data[i11] = array_data[j11];
    array_data[j11] = tmp;
    export(i11);
  }
```

```
    opr int put(i)
      int              i;
    {
      export(array_data[i]);
    }

    opr int get(i)
      int              i;
    {
      export(array_data[i]);
    }

    opr int compare(i1, j1)
      int i1, j1;
    {
      int              a, b;

      a = array_data[i1];
      b = array_data[j1];
      export(a - b);
    }

    {}

  } qsarray(ARRAY_SIZE);
```

```
/*****************************************************************
        Activity sort implements sort-merge algorithm.
*****************************************************************/
act sort(left, right, nz)
     int        left, right, nz;
{
  int              j, k, l, m, ez, temp[ARRAY_SIZE], a, b;
  int              tmp;
  sort             subsort[2];

  ez = right - left;
  m = left + ez / 2;
  if (ez > nz) {
    subsort[0] (left, m, nz);
    subsort[1] (m + 1, right, nz);
    join(subsort[0], subsort[1]);
    j = left;
    k = m + 1;
    l = 0;
     do {
      if ((j <= m) && (k <= right)) {
        tmp = qsarray.compare(j, k);
        if (tmp <= 0) {
          temp[l++] = qsarray.get(j++);
        } else {
          temp[l++] = qsarray.get(k++);
        }.
      }
      if ((k > right) && (j <= m)) {
        temp[l++] = qsarray.get(j++);
      }
      if ((j > m) && (k <= right)) {
        temp[l++] = qsarray.get(k++);
      }
    } while (l <= ez);
    k = left;
    for (l = 0; l <= ez; l++)
      qsarray.put(k++) = temp[l];
    exit();
  }

  j = left;
  k = right;
  do {
    tmp = qsarray.compare(j, ++j);
    if (tmp > 0) {
      qsarray.swap(j--, j);
      if (j > left)
        j--;
    }
  } while (j < right);
};
```

```
init_array(n)
        int                 n;
{
  int               i, data;

  for (i = 0; i < n; i++) {
    scanf("%d", &data);
    qsarray.put(i) = data;
  }
}



print_array(n1, n2)
        int                 n1, n2;
{
  int               i, tmp;

  for (i = n1; i <= n2; i++) {
    tmp = qsarray.get(i);
    printf("%d ", tmp);
  }
  printf("\n");
}



main(argc, argv)
        int                 argc;
        int               **argv;
{
  int               n, nz, data;
  sort              qsort;

  if (argc != 3) {
    fprintf(stderr, "USAGE: sortm size subsize\n");
    exit();
  }
  n = atoi(argv[1]);
  printf("Please enter %d integers\n", n);
  nz = atoi(argv[2]);
  init_array(n);
  printf("\nARRAY:\n");
  print_array(0, n - 1);
  qsort(0, n - 1, nz);
  join(qsort);
  printf("\nRESULT:\n");
  print_array(0, n - 1);
}
```

Appendix VII
Search.mpc

Search is the parallel implementation of a search algorithm and can be called with the command:

```
search processes size key
```

Processes is the number of processes that the program will be allowed to create in order to do the search. Size is the size of the array to be searched. The program will automatically create an array with integer elements ascending from 1 to Size. Key is the number to be searched for and must be within the range of the array. Checking is not conducted in the program for the sake of simplicity and size.

The program behaves like a binary search but differs in that more than one comparison can be made at a time. If three processes are allowed, then three evenly spaced spots in the array are selected and the elements at these positions are compared with the value of the key. The third of the array which must contain the key is then divided with three comparisons, and so on, until the key is hit by a comparison. The program will print the number of the set of comparisons just completed and print the found key when done.

```
/*
 * This is a parallel search algorithm to be executed on an
 * ordered list of elements.
*/

#include <stdio.h>

#define ARRAY_SIZE 10000

frame srchdat(n)
int n;
{
  int array_data[n];
  int proc_flg[32];
  int found;

  opr int put(i)
    int i;
  {
    export(array_data[i]);
  }

  opr int get(i)
    int i;
  {
    export(array_data[i]);
  }
```

```
  opr int set_flg(i)
    int i;
  {
    export(proc_flg[i]);
  }

  opr int read_flg(i)
    int i;
  {
    export(proc_flg[i]);
  }

  opr void set_fnd(i)
    int i;
  {
    found = i;
    export();
  }

  opr int read_fnd()
    {
      export(found);
    }

  {
    found = -1;
  }

} srarray(ARRAY_SIZE);



act compare(procnum,pos,key,n)
int procnum,pos,key,n;
{
  int tmp;

  tmp = srarray.get(pos);
  if ((pos>n) || (tmp>key))
    {
      srarray.set_flg(procnum) = -1;
      exit();
    }
  tmp = srarray.get(pos);
  if (tmp<key)
    {
      srarray.set_flg(procnum) = 1;
      exit();
    }
  srarray.set_flg(procnum) = 0;
  srarray.set_fnd(pos);
};
```

```
init_array(n)
int n;
{
  int i;

  for (i = 1; i < n+1; i++)
    srarray.put(i) = i;
}



main(argc,argv)
int argc;
int **argv;
{
  int n, i, key, tmp;
  int sze, flg, left, right;
  int pss,prc, p[32];
  compare scompare[32];

  if (argc!=4)
    {
      fprintf(stderr,"USAGE: search processes size key\n");
      exit();
    }
  prc = atoi(argv[1]);
  n   = atoi(argv[2]);
  key = atoi(argv[3]);
  init_array(n);
  pss = 1;
  left = 0;
  right = n;
  do
    {
      printf("Pass %d\n",pss++);
      sze = (right - left+1)/(prc+1);
      if (sze == 0)
        sze = 1;
      p[0] = left+sze;
      for(i = 1; i < prc; i++)
        p[i] = p[i-1] + sze;

      for(i = 0; i < prc; i++)
        scompare[i](i,p[i],key,n);

      for(i = 0; i < prc; i++)
        join(scompare[i]);
```

```
        i=0;
        tmp = srarray.read_flg(i);
        while((tmp != -1) && (i < prc))
          {
            i++;
            tmp = srarray.read_flg(i);
          }
        right = left+sze*(i+1);
        left  = left+sze*i;
        flg=0;
        for (i = 0; i < prc; i++)
          {
            tmp = srarray.read_flg(i);
            if (tmp == 0)
              flg = 1;
          }
      } while (flg != 1);

    tmp = srarray.read_fnd();
    printf("%d found at %d.\n",key,tmp);
  }
```

Appendix VIII
Sieve.mpc

Sieve of Erastothenes is an algorithm for extracting the prime numbers from the vector of integers from 1 to N. The basic algorithm is that one activity will start at 1 and compute whether or not an integer is prime. In parallel, other activities will use the results which the first activity has already computed to eliminate other integers which aren't prime.

Sieve can be called with the command:

`sieve numproc list_limit output[y/n]`

where `numproc` is the total number of parallel activities to start, and where `list_limit` is the highest numbered integer to check. The third parameter should be `y` or `n` to indicate whether or not Sieve should output its results or not.

```
#define MAXPROCESSORS 1000
#define LIMIT 100000
#define TRUE 1
#define FALSE 0

#include <math.h>



frame list_array(n)
int n;
{
  char list_data[n];

  opr char put(i)
    int i;
  {
    dsync(list_data[i]) {
      export(list_data[i]);
    }
  }

  opr char get(i)
    int i;
  {
    export(list_data[i]);
  }
```

```
  opr void init(i)
    int i;
  {
    int j;
    for (j=0;j<i;j++)
      list_data[j] = '*';
  }

  {}
} LIST(LIMIT);



frame pt()
{
  int point_data,num;

  opr int test(limit,root1)
    int limit, root1;
  {
    char temp;
    int ti;

    sync(test) {
      do {
        ti = ++point_data;
        if (ti >= limit ||
            num++ >= root1) {
            ti = -1;
            break;
          }
        temp = LIST.get(ti);
      } while(temp != '*') ;
      export(ti);
    }
  }

  { point_data = 1; num = 0;}
} startpoint();
```

```
act slave(processor,limit,root1)
     int processor, limit,root1;
{
  int step_size, place;
  int st_point;
  char dum;

  while(1) {
    st_point = startpoint.test(limit,root1);
    INTEGER_SENSOR(st_point);
    if (st_point == -1){
      exit(1);
    }
    else
      {
        step_size = st_point;
        place =  st_point + step_size;
        while (place < limit) {
          dum = LIST.get(place);
          if (dum != ' ')
            LIST.put(place) = ' ';
          place = place + step_size;
        }
      }
  }
} sieve_slave[MAXPROCESSORS];
```

```
main (argc,argv)
     int  argc;
     char **argv;
{
  int i, processor, limit, rootlimit;
  char cc,ss[32];
  extern double sqrt();

  if (argc < 4){
    printf("Usage sieve numproc list_limit output[y/n]\n");
    exit();
  }
  limit = atoi(argv[2]);
  rootlimit = (int)sqrt((double)limit) + 1;
  LIST.init(limit);

  for(processor = 0; processor < atoi(argv[1]); processor++)
    sieve_slave[processor](processor,limit,rootlimit);
  for(processor = 0; processor < atoi(argv[1]); processor++)
    join(sieve_slave[processor]);

  if (argv[3][0] == 'y'){
    printf("PRIME NUMBERS 1 - %d:\n",limit);
    for(i = 0; i < limit; i++){
      cc = LIST.get(i);
      if (cc == '*'){
        if (i%10 == 0) printf("\n");
        printf("%d ", i);
      }
    }
    printf("\n");
  }
}                                    /* end main */
```

Appendix IX
Mail.mpc

The Mail program is called with the command:

```
mail
```

It simulates a simple mail system by creating three processes, one each for users Mark, Dado, and Nino. Each process allocates a queue in the global mailbox frame for itself. Then they send their own name (either Mark, Dado or Nino) to the other queues in the mailbox. The processes check their own queues then and print the messages they received (the other names). A spooler is also used for the prints to the screen so that characters are printed only once (parallel printing to the screen often results in garbage).

```
/************************************************************

        Example how to use frames and activities

    NOTE-The whole example is synthetic in the sense that
         it is constructed to exhibit the possibilities of
         nesting the frames and operations.(ie. test is
         separated from get and put in the frame que to
         prepare ground for test-and-put and test-and-get
         kinds of operations on the higher hierarchical level)
         There are other ways to build mailboxes, but the code
         below is meant for testing the frames and activities.

************************************************************/
#include <stdio.h>
#include <strings.h>

#define EMPTY           0
#define FULL            1
#define READY           2
#define MATCH           0
#define UNSUCC          -1
#define NAME_LEN        32
#define CUSTOMER_NUMBER 8
#define CUSTOMER_SIZE   256
```

```
frame que (length)
     /***
       This is the template for the shared circular que. Each
       instance will be of a dimension length. Length has to
       be constant at the instantiation time.
       ***/
     int length;
{
  /***
    que_data, read_ptr and write_ptr are shared data
    ***/
  char que_data[length];
  char *read_ptr,*write_ptr;

  opr char get()
    /***
      Get returns character which is on the top of the que.
      To do it one should put in the code yy=xxx.get()
      where yy is any variable, and xxx is a frame instance
      name of a type que.
      ***/

    {
      if((long)read_ptr >= ((long)que_data + length))
        read_ptr = que_data;
      /***
        export points to the part of the operation which will be
        unfolded as a macro exactly at the place of the original
        call inside the statement into the calling code.
        Everything which is above export in the operation will
        be unfolded above the original statement. Simmilar is
        true for the below part.
        ***/
      export(*read_ptr++);
    }
```

```
opr char put()
  /***
    Put puts the data on the top of the que. The sintax
    of the call is xxx.put() = yy, where xxx and yy are
    same as above.
    ***/
  {
    if((long)write_ptr >= (long)(que_data + length))
      write_ptr = que_data;
    export(*write_ptr++);
  }

opr int  test()
  /***
    will test the condition of the que. We don't sync
    because read and send as the higher operations
    combining test get and put will do it
    ***/
  {
    int ret_value;

    switch((int)read_ptr - (int)write_ptr){
    case 0:
      ret_value = EMPTY;
      break;
    case 1:
    case (length - 1):
    case (1 - length):
      ret_value = FULL;
      break;
    default:
      ret_value = READY;
      break;
    }
    export(ret_value);
  }

{
  /***
    This last block without label is frame init section to be
    executed upon invocation. It may be {empty}
    ***/
  write_ptr = que_data;
  read_ptr = que_data;
}

};
```

```
frame mailbox(customer_number,customer_size)
int customer_number;
int customer_size;
{
  /***
    example of the instantiation of previously defined
    frame mailbox_que is a pool of ques to be used in
    a mailbox here has CUSTOMER_NUMBER of ques each of
    them CUSTOMER_SIZE bytes long
    ***/

  que mailbox_que(CUSTOMER_SIZE)[CUSTOMER_NUMBER];
  int read_;

  frame list(list_size)
    /***
      local frame definition example instatiation of list
      which is the name directory of this mailbox is attached
      at the end of the definition itself
      ***/

    int list_size;
  {
    struct {
      char names[NAME_LEN];
    } name_list[list_size];

    opr int  find(name)
      /***
        find the customer by name and return the corresponding id
        ***/
      char *name;
    {
      int ret_value, i;

      ret_value = UNSUCC;
      for(i=0;i<list_size;i++){
        if(strcmp(name_list[i].names,name) == MATCH){
          ret_value = i;
          break;
        }
      }
      export(ret_value);
    }
```

```
  opr int  enlist(name,id)
    /***
      enlist the new customer with given id
      ***/
    char *name;
  int id;
  {
    export(strcpy(name_list[id].names,name));
  }
  /***
    name_list initialization
    ***/
  {
    int i;

    for(i=0;i<list_size;i++)
      name_list[i].names[0]='\0';
  }

} mailbox_name_list(CUSTOMER_SIZE); /* this is the instatiation */
```

```
opr int  send(id,buff,len)
  /***
    send the buffer of the len characters to the
    customer id
    ***/
  int id;
char *buff;
int len;
{

  int i;

  /***
    sync describes the synchronization discipline.
    it will prevent the calling operation to start
    until any of the operations given as parameters
    are in progress.
    NOTE: to exclude mutually each other both parties
    have to call sync on each other.
    One can also call sync on itself.
    ***/
  sync(send){
    int test;

    for(i=0;i<len;i++)  {
      test =  mailbox_que[id].test();
      if(test == FULL){
        printf("sender que full\n");
        break;/* no more space in receiver que -
                 one can use sigkill here to notify sender */
      }
      mailbox_que[id].put() = *(buff + i);
    }
    export(i);
  }
}
```

```
opr int  allocate(customer_name)
  char *customer_name;
{

  int id,find;

  sync(allocate,deallocate){
    find = mailbox_name_list.find(customer_name);

    if(find == UNSUCC){
      find = mailbox_name_list.find("");

      if((id = find) != UNSUCC){
        mailbox_name_list.enlist(customer_name,id);
      } else {
        id = UNSUCC;
      }
    } else {
      id = UNSUCC;
    }
    export(id);
  }
}

opr int  deallocate(id)
  int id;
{
  export(mailbox_name_list.enlist("",id));
}

opr int  locate(name)
  char *name;
{
  export(mailbox_name_list.find(name));
}
```

```
  opr int  read(id,buff,len)
    /***
      reads the message terminated with '\0' from the que
      of the customer id and puts it in the buff of the dimension
      len
      ***/
    int id;
  char *buff;
  int len;
  {
    int i,test;

    i = 0;
    while (i < len) {
      test = mailbox_que[id].test();
      if(test  != EMPTY){
        buff[i] = mailbox_que[id].get();
        if (buff[i++] == '\0'){
          break;
        }
      }
    }
    buff[len-1] ='\0';
    export(i);
  }

  {
    /* no initialization due to the fact that
       list and ques are already initialized */
  }
};
```

```
frame spool()
{
  que internal_que(4096);

  opr int  read(buff,len)
    char *buff;
  int len;
  {
    int i, test;

    i = 0;
    while (i < len) {
      test = internal_que.test();
      if(test != EMPTY){
        buff[i] = internal_que.get();
        if (buff[i++] == '\0'){
          break;
        }
      }
    }
    buff[len-1] ='\0';
    export(i);
  }

  opr int  write(buff)
    char *buff;
  {
    int i,len,test;

    /* NO OTHER WRITE in parallel */
    sync(write){


      len = strlen(buff) + 1;
      for(i=0;i<len;i++) {
        test = internal_que.test();
        if(test == FULL){
          printf("sender que full\n");
          break;/* no more space in receiver que -
                   one can use sigkill here to notify sender */
        }
        internal_que.put() = *(buff + i);
      }
      export(i);
    }
  }
  { }                       /*  NO initialization */
} spool_que();              /* this is the instantiation */
```

```
act sp()
{
  char buff[256];
  int ll;

  while(1){
    ll = spool_que.read(buff,256);
    if (ll > 0){
      if (strcmp(buff,"@@@@@end") == 0)
        break;
      printf("%s",buff);
      fflush(stdout);
    }
  }
} spooler;



main()
{
  /* First we instantiate the frame */
  mailbox mailbox1(CUSTOMER_NUMBER,CUSTOMER_SIZE);

  /* Then we declare local activity user */
  /* It takes three names, and:
     -Takes first name and allocates que in the mailbox;
     -Takes second and third name and locates user id-s;
     -Sends the messages with its name to other two users;
     -Receives two messages in tempbuff1 and tempbuff2;
     */

  char spool_buff[256];
```

```
act user(name1,name2,name3)
  char *name1,*name2,*name3;
{
  char tempbuff1[CUSTOMER_SIZE],tempbuff2[CUSTOMER_SIZE];
  int len1,len2,id1,id2,id3;
  char spool_buff[256];
  strcpy(tempbuff1,name1);
  len1 = strlen(tempbuff1) + 1;
  id1 = mailbox1.allocate(name1);
  if(id1 == UNSUCC){
    printf("no mailbox available\n");
    exit();            /* SOMETHING IS WRONG WITH THE POST OFFICE */
  }
  sprintf(spool_buff,"%s allocated:id1 = %d\n",name1,id1);
  spool_que.write(spool_buff);
  id2 = UNSUCC;
  while(id2  == UNSUCC){
    id2 = mailbox1.locate(name2);
  }
  sprintf(spool_buff,"%s located:id2 = %d,name2 = %s\n",
          name1,id2,name2);
  spool_que.write(spool_buff);

  id3 = UNSUCC;
  while(id3 == UNSUCC){
    id3 = mailbox1.locate(name3);
  }
  sprintf(spool_buff,"%s located:id3 = %d, name3 = %s\n",
          name1,id3,name3);
  spool_que.write(spool_buff);
```

```
    mailbox1.send(id2,tempbuff1,len1);
    sprintf(spool_buff,"%s sent to:%s done\n",name1,name2);
    spool_que.write(spool_buff);
    mailbox1.send(id3,tempbuff1,len1);
    sprintf(spool_buff,"%s sent to:%s done\n",name1,name3);
    spool_que.write(spool_buff);
    len1 = 0;
    while(len1 == 0){
      len1 = mailbox1.read(id1,tempbuff1,64);
    }
    tempbuff1[len1] = '\0';
    sprintf(spool_buff,"%s received:buff = %s,len1 = %d\n",
            name1,tempbuff1,len1);
    spool_que.write(spool_buff);
    len2 = 0;
    while(len2 == 0){
      len2 = mailbox1.read(id1,tempbuff2,64);
    }
    tempbuff2[len2] = '\0';
    sprintf(spool_buff,"%s received:buff = %s,len2 = %d\n",
            name1,tempbuff2,len2);
    spool_que.write(spool_buff);
  };
  user userx[3];
  mailbox mailbox2(CUSTOMER_NUMBER,CUSTOMER_SIZE);
  int i;

  spooler();
  i = 0;

  userx[i]("Mark","Nino","Dado");
  i = 1;

  userx[i]("Nino","Dado","Mark");
  i = 2;

  userx[i]("Dado","Mark","Nino");

  fprintf(stderr,"Joining userx0,1,2...\n");
  join(userx[0], userx[1], userx[2]);
  sprintf(spool_buff,"@@@@@end");
  spool_que.write(spool_buff);
  fprintf(stderr,"join spooler...\n");
  join(spooler);
  /* activity main would exit after sync condition is satisfied */
}
```

Appendix X
Sum.mpc

This is a parallel summation algorithm, which can be called with the command:

sum low high small_enough

where low is the lower bound of the range of integers to sum, and high is the upper bound, and small_enough will determines the amount of parallelism. For example, the range of integers will be recursively subdivided in half until a sub-vector has small_enough elements in it. Please see the program's comments (below) for detail.

```
/*
 * sum will add all the integers in a given range, from MIN to MAX,
 * inclusive. the general algorithm is that the range will be
 * subdivided and recursively summed, and the results will be added
 * together. the user provides a value, SMALL_ENOUGH, which indicates
 * when the subdividing should stop, and the sub-range added with a
 * FOR loop.
 *
 * however, sum knows when there are no more "virtual processes"
 * available to run activities on, and will stop subdividing when this
 * limit is reached. this limit can be changed by using the --P
 * switch to the MPC pre-processor. the default is sixteen (16)
 * virtual processes.
 *
 * this feature is what makes "sum" an interesting example. note that
 * when the program is run in monitoring mode, the sensor collector
 * will use one of the available virtual processes. thus, the
 * behavior of "sum" may change when run in non-monitoring mode. this
 * can be corrected for by using the --P switch to increase the number
 * of virtual processes by one (1) when in monitoring mode.
 */
```

```
#include <stdio.h>



/*
 * the leaves of the decomposition will just add their results to the
 * running total which is kept in "resval" in this shared frame.
*/
frame sum_f()
{
  int resval;

  opr void putsum(val)
    int val;
  {
    sync(get,putsum)
      {
        resval = resval + val;
      }
  }

  opr int get()
    {
      sync(get,putsum)
        {
          export(resval);
        }
    }

  {
    /* initialization */
    resval = 0;
  }
} result();



extern void sum_p();    /* forward declaration */

act sum_a(low,high,smallenough)
     int low,high,smallenough;
{
  sum_p(low,high,smallenough);
};
```

```
void sum_p(low, high, smallenough)
     int low,high,smallenough;
{
  sum_a sub_a;
  int   tmpa, i, tl, th;

  INTEGER_SENSOR(low);
  INTEGER_SENSOR(high);
  if (low == high)
    {
      result.putsum(low);
      return;
    }

  /*
   * pcount        the number of virtual processes which are in use.
   * max_processes the number of virtual processes available total.
   */
  if (workload-> pcount < max_processes)
    {
      if (smallenough == 1)
        {
          /* recursively unroll the rest of the dataset */
          result.putsum(low);
          sub_a(low+1,high,smallenough);
          return;
        }

      if ((high - low + 1) > smallenough)
        {
          /* subdivide, keeping half for ourselves */
          tl = low + ((high-low) / 2) - 1;
          th = low + ((high-low) / 2);
          if (low <= tl && tl <= high)
            sub_a(low,tl,smallenough);
          if (low <= th && th <= high)
            sum_p(th,high,smallenough);
          join(sub_a);
          return;
        }
    }

  /* it is already small enough */
  tmpa = 0;
  for (i = low; i <= high; i++)
    tmpa = tmpa + i;

  result.putsum(tmpa);
}
```

```
main(argc, argv)
  int argc;
  char **argv;
{
  int the_result;

  if (argc  != 4 ||
      (argc == 4 && atoi(argv[1]) >  atoi(argv[2])) ||
      (argc == 4 && atoi(argv[3]) <= 0))
    {
      printf("usage: %s <low> <high> <small_enough>\n",argv[0]);
      printf("\twhere <low> is less or equal to <high>\n");
      printf("\twhere <small_enough> is greater than 0\n");
      exit(0);
    }

  sum_p(atoi(argv[1]),atoi(argv[2]),atoi(argv[3]));

  the_result = result.get();
  printf("%d\n",the_result);
}
```

Appendix XI
Pde.mpc

This example program calculates a PDE in parallel. This algorithm subdivides the grid into subgrids, with a new activity assigned to calculate each subgrid. When the subgrid reaches a user-specified width and height, subdividing is stopped and the PDE is calculated sequentially for that subgrid.

This takes a user-entered square array which holds the initial values for the grid. PDE also takes dimensions which indicate the width and height of the subgrid which is considered to be "small enough" to calculate sequentially.

It is called with a command like:

```
pde w_gridsize h_gridsize w_subgridsize h_subgridsize max_iter
```

where `w_gridsize` and `h_gridsize` are the dimensions of the input grid, where `w_subgridsize` and `h_subgridsize` are the dimensions of the grid which is "small enough". Once PDE is doing its calculations sequentially, it will run trying to converge, or until it has made `max_iter` iterations.

```
#define XMAX 100                /* maximal width of the grid */
#define YMAX 100                /* maximal length of the grid */
#define W 0.5                   /* weight factor */
#define EPSILON 0.1             /* convergence precision */

#include <stdio.h>

frame grid(X, Y)
        int             X, Y;
{
  float            grid_data[X][Y]; /* actual grid data */
  int              conv_flags[X][Y]; /* 1- if difference to
                                      * previous iteration value
                                      * on this grid element is
                                      * less than EPSILON
                                      * 0-otherwise */
  int              pde_control;  /* 1-if everything should finish,
                                  * 0-otherwise */

  opr float data(x, y)
    int x, y;
  {
    export(grid_data[x][y]);
  }

  opr int flag(x, y)
    int x, y;
  {
    export(conv_flags[x][y]);
  }
```

```
    opr int done()
      {
        export(pde_control);
      }

    {      /* initialization of flags */
      int i, j;

      pde_control = 0;
      for (i = 0; i < XMAX; i++)
        for (j = 0; j < YMAX; j++){
          grid_data[i][j] = 0;
          conv_flags[i][j] = 0;
        }
    }
  } pdegrid(XMAX, YMAX);  /* create grid instantiation "pdegrid" */
```

```
act pdecalc(x1, x2, y1, y2, mx, my, limh, limv, maxiter)
int             x1, x2, y1, y2, mx, my, limh, limv, maxiter;
/***

     (0,0)
         *****************
         I               I
         I               I
         I(x1,y1)        I
         I    +++        I
         I    |  |       I
         I    |  |       I
         I    +++        I
         I       (x2,y2) I
         *****************
                      (limh-1,limv-1)

***/
/**
  limh and limv represent the dimensions of the original grid
  before any subdivisions were made, while x1,x2,y1,y2 define
  the subgrid given to this activity
  **/

/**
  mx and my are the user supplied parameters which control the
  process of splitting (ie. when x2-x1 and y2-y1 are smaller than
  mx and my dimensions, splitting process should stop)
  **/

/**
  maxiter is limit on the number of iterations
  **/
{
  pdeproc(x1, x2, y1, y2, mx, my, limh, limv,maxiter);
};
```

```
pdeproc(x1, x2, y1, y2, mx, my, limh, limv,maxiter)
     int x1, x2, y1, y2, mx, my, limh, limv, maxiter;
{
  int             attempt,h,v,left,right,up,down,sv;
  int               ex, ey, i, j, k, nx, ny;
  float             t,tmp,test, temp, a,b,c,d,e;
  pdecalc           subtask;

  ex = x2 - x1 + 1;
  ey = y2 - y1 + 1;

  nx = ex/(2 * mx);
  ny = ey/(2 * my);

  if (ex > mx) {
    /***
      x dimension of the submatrix is larger than mx limit
      (means we have to cut it in two partitions)
      ***/
    if (nx == 0 ) nx = 1;
    subtask (x1,(x1+(nx*mx)-1),y1,y2,mx,my,limh,limv,maxiter);
    pdeproc((x1+(nx*mx)),x2,y1,y2,mx,my,limh,limv,maxiter);
    join(subtask);
    return;
  }
  if (ey > my) {
    /***
      y dimension is larger than my limit
      ***/
    if (ny == 0 ) ny = 1;
    subtask(x1,x2,y1,(y1+(ny*my)-1),mx,my,limh,limv,maxiter);
    pdeproc(x1,x2,(y1+(ny*my)),y2,mx,my,limh,limv,maxiter);
    join(subtask);
    return;
  }
```

```
  /***
    The product submatrix calculated
    ***/
  attempt = 0;
  test = 0;
  do {                                      /* iterate until convergence */
    for (h = x1; h <= x2; h++) {
      for (v = y1; v <= y2; v++) {
        if ((h-1) < 0) left = limh - 1;
        else           left = h - 1;
        if ((h+1) >= limh) right = 0;
        else               right = h + 1;
        if ((v-1) < 0) up = limv - 1;
        else           up = v - 1;
        if ((v+1) >= limv) down = 0;
        else               down = v + 1;
        a = pdegrid.data(h, v);
        b = pdegrid.data(left, v);
        c = pdegrid.data(right, v);
        d = pdegrid.data(h, up);
        e = pdegrid.data(h, down);
        temp = W * a + 0.25 * (1. - W)*(b + c + d + e);
        if (attempt > maxiter){
          pdegrid.flag(h, v) = 1;
        } else {
          if ((tmp - a) > EPSILON ||
              (tmp - a) < (-EPSILON)) {
              pdegrid.flag(h, v) = 0;
            } else {
              pdegrid.flag(h, v) = 1;
            }
        }
        pdegrid.data(h, v) = temp;
      }
    }
    attempt++;
    test = pdegrid.done();
  } while (!test);
}
```

```
void wait_convergence(h, v)
     int h, v;
{
  int i, j, x, y, fl;

  x = h - 1;
  y = v - 1;
  for (i = 0;; (i++, (i == h) ? (i = 0) : i)) {
    for (j = 0;; (j++, (j == v) ? (j = 0) : j)) {
      /*
       * if this guy still doesn't converge set up full
       * cycle marker on it
       */
      fl = pdegrid.flag(i, j);
      if (fl == 0) {
        x = i;
        y = j;
      } else {
        /* full cycle with all flags == 1 ? */
        if (i == x && j == y) { /* YES */
          goto done;
        }
      }
    }
  }
 done:
  /* all flags are 1! Let's finish ! */
  pdegrid.done() = 1;
}



init_grid(h, v)
     int h, v;
     /***
       input grid elements
       ***/
{
  int             x, y;
  float           tmp;

  printf("enter grid row by row\n");
  printf("row separator is CR\n");
  for (y = 0; y < v; y++) {
    for (x = 0; x < h; x++) {
      scanf("%f", &tmp);
      pdegrid.data(x, y) = tmp;
    }
    while (getchar() != '\n');
  }
}
```

```
print_result(h, v)
     int h, v;
{
  int              x, y;
  float            tmp;

  printf("\nresult:\n");
  for (y = 0; y < v; y++) {
    for (x = 0; x < h; x++) {
      tmp = pdegrid.data(x, y);
      printf("%e ", tmp);
    }
    printf("\n");
  }
}
```

```
main(argc, argv)
     int         argc;
     char        *argv[];
{
  int              xs, ys, mx, my, mi;
  pdecalc          calc;

  if (argc != 6) {
    fprintf(stderr, "Usage: pde xsize ysize mx my mi\n");
    exit();
  }

  xs = atoi(argv[1]);   /* x-dim. of the grid */
  ys = atoi(argv[2]);   /* y-dim. of the grid */
  mx = atoi(argv[3]);   /* split dim. of the grid ie. splitting */
  my = atoi(argv[4]);   /* process will stop when the corespon. */
  mi = atoi(argv[5]);   /* dim. of resultant subgrid is less than */
                        /* my or my respectively */

  if ((xs > XMAX - 1) || (ys > YMAX - 1)) {
    fprintf(stderr, "grid too big: max(%d,%d)\n", XMAX, YMAX);
    exit();
  }

  init_grid(xs, ys);

  /* start the initial activity */
  calc(0, xs - 1, 0, ys - 1, mx, my, xs, ys,mi);

  wait_convergence(xs, ys);      /* master is checking on convergence
                                  * in parallel while waiting the
                                  * activities and teams to do the
                                  * calculations. This is simple
                                  * function call due to the fact that
                                  * master process has nothing to do
                                  * in the meantime anyway */

  join(calc);                    /* DONE join activity */

  print_result(xs, ys);
}
```

Appendix XII
MPC Grammar

Much of the grammar in this section was taken from *"A C reference Manual"* [Harbison and Steele 84]. The additional constructs unique to MPC can be found at the end of this section. Also refer to Chapter .

program ::= { *top-level-dec* }*

top-level-dec ::= *top-level-data-dec*
 | *top-level-function-dec*

top-level-data-dec ::= { *type-class-spec* }* { *init-dcltr* #',' }* ';'
 | *frame-spec* { *frame-dcltr* #',' }* ';'
 | *activity-spec* { *activity-dcltr* #',' }* ';'
 | *name-type-def* ';'
 | *type-def-spec* ';'

top-level-function-dec ::= { *type-class-spec* }* { *param-dec* }* *compound-stmt*

local-data-dec ::= { *type-class-spec* }+ { *init-dcltr* #',' }* ';'
 | *frame-spec* { *frame-dcltr* #',' }* ';'

activity-spec { *activity-dcltr* #',' }* ';'
 | *name-type-def* ';'
 | *type-def-spec* ';'

name-type-def ::= `typedef` { *type-spec* }+ { *p3-dcltr* #',' }+
 | `typedef` *identifier* { *p3-dcltr* #',' }+

type-name ::= { '*' }* *identifier* { '(' *list-expression* ')' }? { '[' *list-expression* ']' }? { '=' *init-expression* }?

type-def-spec ::= *identifier* { *type-name* #',' }+

parameter-dec ::= *dec-spec* { *p3-dcltr* #',' }* ';'

type-name-dec ::= { *type-class-spec* }+ { *p3-abs-dcltr* }?

formal-dec ::= { *type-class-spec* }* *p3-dcltr*

type-class-spec ::= *standard-class*
| *type-spec*

standard-class ::= `auto`
| `static`
| `extern`
| `register`

type-spec ::= *standard-type*
| *structure-spec*
| *enum-spec*

standard-type ::= `char`
| `float`
| `double`
| `int`
| `short`
| `long`
| `unsigned`
| `void`

structure-spec ::= `struct` *identifier*
| `union` *identifier*
| `struct` { *identifier* }? '{' { *structure-dec* }* '}'
| `union` { *identifier* }? '{' { *structure-dec* }* '}'

structure-dec ::= { *type-class-spec* }+ ';'

structure-dcltr ::= *p3-dcltr*
| { *p3-dcltr* }? ':' *expression*

enum-dec ::= '{' { *enum-dcltr* #',' }+ { ',' }? '}'

enum-spec ::= `enum` *identifier*
| `enum` { *identifier* }? *enum-dec*

enum-dcltr ::= *identifier* { '=' *expression* }?

p1-dcltr ::= *identifier*
| '(' *p3-dcltr* ')'

p2-dcltr ::= *p1-dcltr*
| *p2-dcltr* '(' { *formal-dec* #',' }* ')'
| *p2-dcltr* '[' *list-expression* ']'

p3-dcltr ::= *p2-dcltr*
| '*' *p3-dcltr*

init-dcltr ::= *p3-dcltr* { '=' *init-expression* }?

init-expression ::= *expression*
| '{' { *init-expression* #',' }+ { ',' }? '}'

p2-abs-dcltr ::= *p1-abs-dcltr*
| { *p2-abs-dcltr* }? '(' ')'
| { *p2-abs-dcltr* }? '[' *list-expression* ']'

p3-abs-dcltr ::= *p2-abs-dcltr*
| '*' { *p3-abs-dcltr* }?

compound-stmt ::= '{' { *dec-or-stmt* }* '}'

dec-or-stmt ::= *local-data-dec*
| *statement*

basic-stmt ::= *e-stmt*
| *compound-stmt*
| *do-stmt*
| *break-stmt*
| *continue-stmt*
| *return-stmt*
| *goto-stmt*
| *sync-stmt*
| *dsync-stmt*
| *join-stmt*
| *detach-stmt*

balanced-stmt ::= *basic-stmt*
| *balanced-while*
| *balanced-for*
| *balanced-ifelse*
| *balanced-switch*

| *label unbalanced-stmt*

unbalanced-stmt ::= *unbalanced-while*
| *unbalanced-for*
| *unbalanced-if*
| *unbalanced-ifelse*
| *unbalanced-switch*
| *label unbalanced-stmt*

balanced-ifelse ::= `if` '(' *list-expression* ')' *balanced-stmt* `else` *balanced-stmt*

unbalanced-ifelse ::= `if` '(' *list-expression* ')' *balanced-stmt* `else` *unbalanced-stmt*

unbalanced-if ::= `if` '(' *list-expression* ')' *statement*

statement ::= *balanced-stmt*
| *unbalanced-stmt*

e-stmt ::= *list-expression* ';'

balanced-while ::= `while` '(' *list-expression* ')' *balanced-stmt*

unbalanced-while ::= `while` '(' *list-expression* ')' *unbalanced-stmt*

do-stmt ::= `do` *stmt* `while` '(' *list-expression* ')' ';'

balanced-for ::= `for` '(' *list-expression* ';' *list-expression* ';' *list-expression* ')' *balanced-stmt*

unbalanced-for ::= `for` '(' *list-expression* ';' *list-expression* ';' *list-expression* ')' *unbalanced-stmt*

balanced-switch ::= `switch` '(' *list-expression* ')' *balanced-expression*

unbalanced-switch ::= `switch` '(' *list-expression* ')' *unbalanced-expression*

break-stmt ::= `break` ';'

continue-stmt ::= `continue` ';'

return-stmt ::= `return` '(' *list-expression* ')' ';'

goto-stmt ::= `goto` *identifier* ';'

label ::= *name-label*
| *case-label*
| *default-label*

name-label ::= *identifier* ':'

case-label ::= `case` *expression* ':'

default-label ::= `default` ':'

literal ::= *integer*
| *float*
| *character*
| *string*

primary-p1-expression ::= *identifier*
| *literal*
| '(' *expression* ')'
| `sizeof` '(' *type-name-dec* ')'

primary-p2-expression ::= *primary-p1-expression*
| *primary-p2-expression* '[' *list-expression* ']'
| *primary-p2-expression* '(' *list-expression* ')'
| *primary-p2-expression* '.' *identifier*
| *primary-p2-expression* '->' *identifier*

postfix-expression ::= *primary-p2-expression*
| *postfix-expression pre-postfix-operator*

pre-postfix-operator ::= '++'
| '--'

prefix-expression ::= *postfix-expression*
| `sizeof` *prefix-expression*
| *pre-postfix-operator cast-expression*
| '*' *cast-expression*
| '&&' *cast-expression*
| *negation-operator cast-expression*

negation-operator ::= '-'
| '!'
| '~'

cast-expression ::= *prefix-expression*
| '(' *type-name-dec* ')' *cast-expression*

multiply-operation-expression ::= *cast-expression*
| *multiply-operation-expression multiply-operator cast-expression*

multiply-operator ::= '*'
| '/'
| '%'

addition-operation-expression ::= { *addition-operation-expression addition-operator* }?
multiply-operation-expression

addition-operator ::= '+'
| '-'

shift-operation-expression ::= { *shift-operation-expression shift-operator* }? *addition-operation-expression*

shift-operation ::= '<<'
| '>>'

relation-operation-expression ::= { *relation-operation-expression relational-operator* }?
shift-operation-expression

relational-operator ::= '<'
| '>'
| '<='
| '>='

equality-operation-expression ::= { *equality-operation-expression equality-operator* }?
relation-operation-expression

equality-operator ::= '=='
| '!='

bitand-operation-expression ::= { *bitand-operation-expression* '&' }? *equality-operation-expression*

bitxor-operation-expression ::= { *bitxor-operation-expression* '^' }? *equality-operation-expression*

bitor-operation-expression ::= { *bitor-operation-expression* '|' }? *equality-operation-expression*

and-operation-expression ::= { *and-operation-expression* '&&' }? *bitor-operation-expression*

or-operation-expression ::= { *or-operation-expression* '||' }? *and-operation-expression*

conditional-expression ::= *or-operation-expression* { '?' *list-expression* ':' *conditional-expression* }?

expression ::= *conditional-expression* { *assignment-operator expression* }?

assignment-operator ::= '='
| '+='

| '-='
| '*='
| '/='
| '%='
| '>>='
| '<<='
| '&='
| '^='
| '|='

list-expression ::= { expression #',' }*

frame-spec ::= frame-tag-dcltr { parameter-dec }* '{' frame-dec '}' ';'

frame-dec ::= { local-data-dec }* { frame-operation }* frame-initialization

frame-operation ::= `opr` { type-class-spec }* (operation-name) { parameter-dec }* operation-body

frame-dcltr ::= identifier { '{' list-expression '}' }? { '[' list-expression ']' }?

frame-tag-dcltr ::= `frame` identifier '(' { formal-dec #',' }* ')'

operation-body ::= '{' local-data-dec { statement }* export-statement { statement }* '}'

export-statement ::= `export` '(' list-expression ')' operation-name ::= identifier '(' { formal-dec #',' }* ')'

frame-initialization ::= compound-stmt

sync-stmt ::= `sync` '(' list-expression ')' compound-stmt

dsync-stmt ::= `dsync` '(' list-expression ')' compound-stmt

join-stmt ::= `join` '(' list-expression ')' ';'

detach-stmt ::= `detach` '(' list-expression ')' ';'

activity-tag-dcltr ::= `act` identifier '(' { formal-dec #',' }* ')'

activity-dcltr ::= identifier { '[' list-expression ']' }?

activity-spec ::= activity-tag-dcltr { parameter-dec }* compound-stmt ';'

References

[Gregoretti 85] Francesco Gregoretti, Zary Segall.
Programming for Observability Support in a Parallel Programming Environment.
Technical Report CMU-CS-85-176, Computer Science Department, Carnegie Mellon University, November, 1985.

[Harbison and Steele 84] S.P.Harbison, G.L. Steele.
A C Reference Manual.
Prentice Hall, Englewood Clifs, NJ 07632, 1984.

[Segall 85] Zary Segall, Larry Rudolph.
PIE - A Programming and Instrumentation Environment for Parallel Processing.
Technical Report CMU-CS-85-128, Computer Science Department, Carnegie Mellon University, April, 1985.

[Snodgrass 82] Richard Snodgrass.
Monitoring Distributed Systems: A Relational Approach.
PhD thesis, Department of Computer Science, Carnegie Mellon University, December, 1982.

[Vrsalovic 84] D. Vrsalovic, D. Siewiorek, Z. Segall, E. Gehringer.
Performance Prediction and Calibration for a Class of Multiprocessor Systems.
Technical Report, Department of Computer Science, Carnegie Mellon University, August, 1984.