MPC - Multiprocessor C Language for Consistent Abstract Shared Data Type Paradigms

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

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dv@cs.cmu.edu
(412) 268-6426

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, Consistent Abstract Shared Data Type Implementation Machine (CASDTIM), and an analytical model for predicting performance of MPC programs using the CASDTIM is presented. The analytic model is then shown to be in close agreement with the measurements of an actual MPC program executing on a commercially available multiprocessor.

Introduction

Modern 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 that the use of parallelism brings increased system complexity. Conventional system design tools tend to cope with increased complexity by imposing a layered hierarchy of abstractions. Intensive use of communication and synchronization resources 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: delays due to contention on common resources, delays due to synchronization overhead, increased load due to unfavorable parallel decomposition, and 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 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 programmers have produced their best design, the role of minimizing remaining performance degradations falls upon the run-time support system. The PIE project (Performance Efficient Parallel/Distributed Programming and Instrumentation Environment), as depicted in Figure 1 supports the entire design process from modeling (e.g. prevention), to monitoring (e.g. bottleneck detection), to run-time (e.g. avoidance).

PIE [5] 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 virtual structure is available during program development time when the abstractions are required to assist in understanding complexity. By run-time, however, the structure has been flattened and removed yielding higher performance parallel programs.
In PIE, synchronization and communication functions are described by high-level primitives. These are translated by the MPC pre-processor into runtime primitives which are optimized for the target machine's architecture. In this way, MPC provides help in reducing the overhead of all four forms of performance degradation described above.

PIE also embraces the concept of "programming for observability" [3] in which users make use of visual tools to aid in the development, testing, and debugging of their application. During development, the PIE system incrementally builds a view of the user program's semantic 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 extra information gleaned from the visual displays will help the user to think more clearly about the program's behavior.

The present PIE environment consists of several components:

- PERMOD [6] is the modeling tool which provides performance prediction in the early design stages of parallel systems.
- MPC (Multiprocessor C) [7] is a C preprocessor that converts special MP (Multi-Processor) language constructs into C program syntax. It implements the "Consistent abstract shared data type implementation machine" (CASDTIM). Despite the fact that the target machine could be of a different architecture, MPC provides the CASDTIM model to the user via synchronization, and shared memory constructs.
- 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 development time data about the target program and assists in instrumenting it for the purpose of run-time monitoring.
- PIEscope allows all the development and 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 sections introduce only the MPC and PERMOD portions of PIE. However, examples throughout the paper are illustrated by the graphical outputs from PIEscope.

**MPC - Multiprocessor C**

The previous section 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.

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.

The current version of MPC supports the Consistent Abstract Shared Data Type Implementation Machine or CASDTIM. Implementation machines differ in the ways synchronization and communication is handled. MPC exports CASDTIM to users via 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 mutual exclusion for specific parts of data in frames.
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.
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. MPC also provides constructs for joining with and detaching from activities.

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 for join and detach.

As was stated, activities are similar to functions. The differences being that they run in parallel with 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 (viz), 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 its 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 it is known that no activity will attempt to join with the completed activity, a detach statement is included in the code after the activity has been invoked. 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) and may appear as below (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>):

\[
\begin{align*}
\text{activity-spec} & : = & \text{activity-tag-dcltr} \{ \text{parameter-dec} \} \ast \text{compound-stmt} \{ \text{activity-dcltr} \} \ast \text{'} ; \text{'} ; \text{'} \\
\text{activity-tag-dcltr} & : = & \text{act} \text{ identifier} \{ \text{'} \{ \text{formal-dec} \} \} \text{' } \ast \text{'} \ast \text{'} \ast \text{'} \ast \text{'} \\
\end{align*}
\]

\[1\text{Note that the procedure main is also considered an activity.}\]

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

\[
\text{activity-dcltr} : = \begin{cases} & \text{activity-tag-dcltr} \{ \text{list-expression} \} \ast \\
& \text{activity-tag-dcltr} \{ \text{activity-instance} \} \ast \text{'} \ast \text{'} ; \text{'} \\
& \text{activity-tag-dcltr} \{ \text{activity-instance} \} \ast \text{'} \ast \text{'} \ast \text{'} \ast \text{'} \\
\end{cases}
\]

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

\[
\begin{align*}
\text{join-statement} & : = & \text{join} \{ \text{activity-instance} \} \ast \text{'} \ast \text{'} ; \\
\text{detach-statement} & : = & \text{detach} \{ \text{activity-instance} \} \ast \text{'} \ast \text{'} \ast \text{'} \ast \text{'} \ast \text{'} \\
\end{align*}
\]

Example: Matrix Multiplication

The example in Figure 2 is an activity definition which includes instantiations of itself, and calls to the instantiations. This activity performs parallel matrix multiplication by dividing 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, the line defines this activity with the name multiply. It also shows that activities of this type require 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.

The activity is passed two variables, max and my representing limits placed on the granularity of how small the matrix can be divided. 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 further divide the submatrix, 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 max by my), the do_mult routine is called to calculate the product of the resultant matrix delimited by x1, x2, y1, and y2.

Hints on the efficient use of activities

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 as in the following example:

\[
\begin{align*}
\text{some_act my_activity[N];} \\
\text{for (i=0; i < N; i++)} \\
\text{\{ my_activity[i](p1, p2, \ldots, pN); } \\
\text{\} for (i=0; i < N; i++)} \\
\text{\{ join(my_activity[i]); } \\
\text{\}
\end{align*}
\]

It is often very useful to pass index i as a parameter to the activity so the functionality can vary at runtime.
act multiply(x1, x2, y1, y2, mx, my, sz)
/**
x1, x2
y1, y2
sz - the original
* * matrix size
* * mx and my - the desired
**
int x1, x2, y1, y2, mx, my, sz;
{
    int ex, ey, i, j, k;
    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 (ex > ey) {
        if (ex > mx) {
            /* cut along x dim. */
            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. */
            
        } else if (ey >= ex) {
            if (ey > my) {
                
            } else if (ex > mx) {
                
            } else {
                /* no more children - do it! */
                do_mult(x1, x2, y1, y2, sz);
            }
        }
    } else { /* no more children - do it! */
        do_mult(x1, x2, y1, y2, sz);
    }
}
*/
}
Figure 2: Parallel matrix multiplication

A much more performance-efficient way to spawn N parallel activities is to use recursion. 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(. . .)
    
    
    do_some_work();

    bar(. . .); /* This will start a activity of type foo */

    do_some_work();

    join(bar);
}

The increased efficiency comes from the fact each spawned activity can spawn more activities thus parallelizing the entire startup process.

Keeping this in mind, let us revisit the matrix multiplication example (in Figure 2). 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. In order to achieve this, the matrix multiplication activity from the previous example should be rewritten as in Figure 3.

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;

    
    
    multiply subtask;

    
    
    /* try to cut the longer side */
    if (ex > ey) {
        if (ex > mx) {
            /* cut along x dim. */
            
        } else if (ey > my) {
            /* cut along y dim. */
            
        } else if (ey >= ex) {
            if (ey > my) {
                
            } else if (ex > mx) {
                
            } else {
                /* no more children - do it! */
                do_mult(x1, x2, y1, y2, sz);
            }
        }
    } else {
        /* no more children - do it! */
        do_mult(x1, x2, y1, y2, sz);
    }
}

Figure 3: Improved Matrix Multiplication in MPC

Both versions of matrix multiplication were executed using identical data and produced PIEScope timelines shown in Figure 4. Note the difference in both the time required to execute the two algorithms 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.
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 with the operations put and get. MPC also provides synchronization support for these operations (both data and control-flow synchronization is available).

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 usables 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 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 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 to contain the result if it could be generated in one of several branches of a condition. In addition, it means that export statements can be unfolded as the left hand side or the right hand side of expressions only. That is, export statements cannot appear as arguments to procedures or in conditional clauses. 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.

Frame Syntax

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

Frame declarations appear as in Figure 4 (note that, as with activities, one may declare instances of a frame as part of its declaration). All syntax forms are given in BNF.

One example of a frame definition is given in Figure 6. 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).

Synchronization within a frame

The above example is fine when you know that the users of a particular instance 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 form of mutual exclusion must be specified on the data and operations of a frame.

The sync and dsync statements allow synchronization of frame operations that are performed in parallel. In

---

2See next section for a description of the frame syntax
while the code in the block is executed. To execute a
sync statement to the end of the sync statement's reach
where,
other words, since frame operations perform actions on
the definition of a frame operation. A sync statement
precedes a block of critical code that stretches from the
sync statement to the end of the sync statement's 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
func statement in
sync-statement
( or vector thereof) is:
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 '}'

Figure 5: BNF Definition of the Frame Syntax

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.

Sync. Sync statements can be included only inside
the definition of a frame operation. A sync statement
precedes a block of critical code that stretches from the
sync statement to the end of the sync statement's 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

Figure 6: A simple frame
sync statement is to preclude execution of the frame
operations named in the parameter list. A frame opera-
tion 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 state-
ment, 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 con-
dition (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 matrix(rank)
int rank;
{
float mat[rank][rank];
opr float get(i,j)
int i, j;
{ export(mat[i][j]); }
opr void put(i,j,val)
int i, j, float val;
{ mat[i][j] = val; }
{
bzero(mat,
siziof(float) *rank*rank);
}
a(5), b(5)[5][5];

Figure 7: Frame skeleton with sync statements

Let's examine some hypothetical cases using the
frame shown in Figure 7. 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 7 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 which will be automatically generated in future versions of MPC preprocessors.

**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, and 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 expressions which will be evaluated at the runtime as their indices (e.g., dsync(a[i][j]) is legal if i will be calculated at runtime prior to the time dsync is executed).

**Synchronization example.** Let us return to the example in Figure 6 to see how the contention problem might be solved using the MPC synchronization constructs. Situations to be avoided include two parallel activities either writing or reading and writing at the same time. Thus multiple activities reading is not a problem as reading is not a destructive operation. So, a straightforward approach might produce the code in Figure 8:

```c
frame matrix(rank)
int rank;
{
  float mat[rank][rank];
  opr float get(i, j)
  int i, j;
  sync (put) {
    export(mat[i][j]);
  }
  opr void put(i, j, val)
    int i, j; float val;
    sync (put, get) {
      mat[i][j] = val;
    }
    bzero(mat, sizeof(float)*rank*rank);
}
```

**Figure 8:** A MPC implementation of a shared matrix

The code segment in Figure 8 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 the code where sync statements are replaced by dsync(mat[i][j]).

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 potential deadlock situations.

**Dynamic Frames.** Frames behave like static variables (i.e., 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 strategy of specifying only a pointer to global data, which can then be allocated at run-time.

**Modeling Performance in Shared Data Systems**

PERMOD, a model for predicting multiprocessor performance of algorithms composed of repeated iterations, called application cycles, is derived in [6]. This section will summarize the model. Each application cycle consists of some amount of access to global data and some amount of local processing. The application cycles may be synchronous or asynchronous, and the processors may or may not incur waiting time, depending on the relationship between the access time and processing time. Unlike a standard queuing or simulation model, PERMOD does not account for randomness in loop iterations. It may therefore be unsuitable for modeling applications where unpredictable branching plays a major role. On the other hand, PERMOD is computationally simpler than many queuing or simulation models, and may thus offer a more tractable means of dealing with complex but regular computations. It is also important to point out that the PERMOD results are statistical; it may thus offer a more tractable means of dealing with complex but regular computations. It is also important to point out that the PERMOD results represent both upper (asynchronous) and lower (synchronous) bounds on performance while other models merely yield statistical averages.

The model assumes the CASDTIM where access time to shared data is independent of the location of the processor. Each iteration is made up of a single period of processing, and a single period of access to global memory (access to local memory is charged to processing time rather than being modeled separately). Due to the large number of iterations, start-up transients are neglected.

The model yields speed-up over a uniprocessor system for the synchronous case.
\[ S = \frac{f_p f_f (1 + X)}{N f_p + X f_f} \]

and the asynchronous case:
\[ S = \min \left\{ \frac{1 + X}{f_p f_p + X f_f}, \frac{1 + X}{N} \right\} \]

where the variables are defined as:
- \( N \) - number of parallel activities;
- \( X \) - local processing to global communication ratio for the overall problem;
- \( f_p \) - local processing decomposition function;
- \( f_f \) - global communication decomposition function;

**An Example: Matrix Multiplication**

In order to measure speedup, we compare the performance of a particular decomposition to the performance of a reference uniprocessor implementation of the same algorithm. Strictly speaking, a uniprocessor implementation need not have any global data—all the data can simply be local to the single processor. For the sake of comparison, however, we will assume that the uniprocessor implementation uses exactly as much global data as its multiprocessor counterparts. In other words, to obtain the number of global references by the uniprocessor implementation, we derive an equation for the number of references by a multiprocessor implementation, and then simply set \( N \) to 1.

Regardless of whether data is local or global, it takes time to access it. Collectively, the time to perform the global data accesses constitutes \( t_g \). The time to perform the local accesses is included in the processing time \( t_p \).

When local copies are substituted for global data, \( t_g \) increases for two reasons: (1) it takes processing time to copy the data, and (2) the time for accessing the data is now charged to \( t_g \) instead of \( t_p \). We will assume that the cost \( t_g \) is negligible. Our development will employ the following notation:
- Greek letters will be used to refer to global variables.
- The symbol "\( i=j,k \)" represents the serial iterations for \( i \) from \( j \) to \( k \) by 1.
- The symbol "\( \leftarrow \)" means assignment.
- The symbol "\( \leftarrow \)" indicates that values are copied in one direction and later copied back.

**Local vs. Global Shared Data Copies**

Here we analyze the decomposition of a parallel algorithm for matrix multiplication. For \( M \times M \) matrices \( A, B \) and \( C \), the solution of the matrix equation \( C = A \times B \) can be decomposed into \( N \) processes, each assigned to a submatrix of dimension \( (M/\sqrt{N}) \times (M/\sqrt{N}) \). We can consider the submatrices themselves as arranged into \( m_1 \) rows and \( m_2 \) columns, and let \( C_{RS} \) be the \( (R,S) \)th submatrix in the \( R \)th row. In other words, this means that

\[ m_i = \frac{M}{\sqrt{N}} \]

\[ C_{RS} = C_{((R-1)m_1+i),((S-1)m_2+j)} \]

Now let the number of submatrices along one dimension of the original matrix be: \( m_2 = \sqrt{N} \). Let \( A_{RS} \) and \( B_{RS} \) denote the \( (R,S) \)th submatrices of \( A \) and \( B \) respectively, and assume that each process does an equal amount of work. Then the work done by an individual process is:

\[ I_{R=1,m_1 \cdot J=1,m_2} \sum_{i=1}^{m_1} \sum_{j=1}^{m_2} A_{RS} \times B_{RS} \]

This equation says that the \( (R,S) \)th process consists of a doubly nested loop that iterates from 1 to \( m_1 \). Each iteration computes a single element of the \( (R,S) \)th result submatrix. The \( (i,j) \)th element of this submatrix is computed by summing \( \sqrt{N} \) dot products of the \( i \)th row of a submatrix of \( A \) with the \( j \)th column of a submatrix of \( B \).

If local copies are used, the \( (R,S) \)th process must copy the submatrices in the \( R \)th row and the \( S \)th column, but other than that, there are no global accesses. Let us denote the submatrix copies by lowercase letters, e.g., \( a_{RS} \). The following copying work is done:

\[ I_{R=1,m_1 \cdot J=1,m_2} \sum_{i=1}^{m_1} \sum_{j=1}^{m_2} A_{RS} \leftrightarrow a_{RS} \]

Let
- \( T_1 \) be the processing time for one iteration without local copies (that is, the time for one iteration in (4)),
- \( T_2 \) be the time to perform one global access, exclusive of waiting time, and
- \( T_3 \) be the processing time for copying a single global matrix element to or from global memory (that is, the time for one iteration in (5)).

Then we can derive the processing and access times \( (t_p, t_g) \) for an individual process by multiplying the number of iterations by the time needed for processing and accesses, respectively, within an iteration.

**Without local copies:**

\[ t_p = (m_1 m_2 m_i) T_1 = \frac{M^3 T_1}{N} \]

\[ t_g = (m_1 m_2 m_i) T_3 = \frac{M^3 T_3}{N} \]

---

*Exhaustive definitions of these parameters can be found in [6]*
Hence,

\[ T_p = M^2 T_1, \quad T_a = 2M^2 T_2, \quad X = \frac{T_1}{2T_2} \]  
\[ f_p = N, \quad f_a = N \]  

With local copies:

\[ T_p = \frac{M^2 T_1}{N} \]  
\[ t_a = (\sqrt{N} m, m) 2T_2 = 2M^2 \frac{T_2}{\sqrt{N}} \]  
\[ T_p = M^2 T_1 + 2M^2 T_3, \quad T_a = 2M^2 T_2 \]  
\[ X = \frac{MT_1 + 2T_3}{2T_2} \]  
\[ f_p = \frac{(MT_1 + 2T_3)N}{MT_1 + 2T_3} \]  
\[ f_a = \sqrt{N} \]  

More Complex Workloads

One should never expect that real workloads will consist of only one group of activities executing only one set of identical iterations. Even such a simple example as the matrix multiplication outlined in Figure 2 has two sequential (i.e. prologue and epilogue) and one parallel (i.e. multiplication) part.

In order to be able to predict the performance of a complete workload let us take a closer look at the speedup of the combination of two groups of parallel activities. We shall assume that the speedup prediction for each such group can be made by using PERMOD. Now let us define the following variables:

- \( S_x \) - speedup of the activity group \( x \);
- \( T_x \) - overall useful work (measured in some time units) done by the group \( x \);
- \( t_x \) - time needed for group \( x \) to process the work \( T_x \) in parallel.

These three values are related by 12.

\[ S_x = \frac{T_x}{t_x} \]  

If a workload consists of two groups of activities, executed in sequential order, the composite speedup will be defined as the ratio between the overall useful work done and the time needed to do it:

\[ S = \frac{T_1 + T_2}{t_1 + t_2} = \frac{(1 + K)S_2 S_1}{S_2 + K S_1} \]  

where \( K = T_2/T_1 \) is a constant which represents the ratio of the amounts of useful work done by each group. In order to find the optimal speedup we use the first derivative of the equation 13.

\[ \frac{dS}{dK} = S_1 S_2 (S_2 + KS_1) - (1 + K)S_1 (S_2 + KS_1)^2 = 0 \]  

Solution of the equation 14 shows that the optimal speedup is reached for:

\[ S_1 = S_2 \]  

One could assume that one of two workload arts is a sequential one with Speedup of 1.

By substituting the value of 1 for \( S_1 \) in equation 13 we get:

\[ S = \frac{(1 + K)N}{N + K} \]  

and finally, by letting \( S_2 \) be equal to \( N \), we get a special case:

\[ S = \frac{(1 + K)N}{N + K} \]  

better known as Amdahl's Law [1]. However, in most of the practical cases some work will not be infinitely parallelizable and \( S_2 \) will be sublinear.

Two extreme cases for \( K \to 0 \) and \( K \to \infty \) are of theoretical interest because they represent speedup boundary values.

\[ \lim_{K \to 0} S = S_2 \] and \[ \lim_{K \to \infty} S = 1 \]

Some Experimental Results

The matrix multiplication algorithm given in Figure 3 was compiled by the MPC preprocessor and executed on a 16 processor Encore Multimax. Since Multimax is a time sharing multiprocessor with active users, the matrix multiply was repeated ten times to obtain a statistical average execution time. The execution times were measured for one to twelve parallel activities and a range of matrix sizes.

Measured execution times are presented in Figure 9. One can immediately see that performance degradation is highest at the lowest granularity of computation (i.e. smaller size of matrix \( M \)). The Encore Multimax employs large caches, and hence, exhibits decomposition functions close to those derived for making local copies (equation 9). Due to the fact that, even for the smallest matrix size, the processing to access ratio is well over the maximal number of processors used (i.e. 12), PERMOD suggests that there will be no degradation due to the contention, synchronization, or unfavorable decomposition for any number of processors in the range of one to twelve.

The reason is that this model takes into account only three of the four performance degradation sources. If we add the influence of an imbalanced workload as developed earlier, we get the predicted values as depicted in Figure 9. To keep the example simple we modeled the test example as if it consisted of two parts: a sequential one needed to start/join parallel activities, and the parallel one actually performing the multiplication. The value of \( K \) was calculated by fitting the

\[ \text{Predicted values are shown as dotted lines and measured values as solid ones.} \]
Conclusions

A machine independent parallel language, MPC, has been defined, a C language preprocessor has been implemented, several algorithms encoded in MPC and run time profiles generated. A simple analytical model guides the programmer in terms of the expected speedup as a function of the number of processors. The MPC and PIE monitoring software has proven to be effective for identifying bottlenecks in parallel programs, and represents a major step towards realizing the performance potential of parallel processors.

Present experience with MPC proves that it is possible to have a structured high level programming environment and still avoid a lot of performance degradations at the object code level. We have successfully written several test applications in MPC. One of them, matrix multiplication, is presented in this paper. We also had a chance to analyze several other production applications written in MPC [4,2]. In all these cases there was no significant overhead observed in the resultant C code, compared to the C code produced by hand.

Our results show that load imbalance is the major source of performance degradation in modern tightly coupled multiprocessor systems, with a range of 10-20 processors. This is especially true if one doesn’t use a "bare machine" but writes applications in a higher level parallel language under a UNIX-like operating system. The model presented here accurately describes the execution of a complex workload consisting of a number of components of various granularity, computational intensity, and potential parallelism. We show that, for one special case, the results obtained from this model are equal to those obtained by the widely known Amdahl’s law. The measured and predicted results for the matrix multiplication example and the specific implementation in MPC show a difference which is within 5%.

References