Explicit Parallel Programming: User's Guide

By Jim Gamble and Calvin J. Ribbens

TR 91-19

Explicit Parallel Programming: User's Guide*

Jim Gamble† and Calvin J. Ribbens
Department of Computer Science
Virginia Polytechnic Institute & State University

July 23, 1991

Abstract

The Explicit Parallel Programming (EPP) language is defined and illustrated with several examples. EPP is a prototype implementation of a language for writing parallel programs for shared memory multiprocessors. EPP may be viewed as a coordination language, since it is used to define the sequencing or ordering of various tasks, while the tasks themselves are defined in some other compilable language. The EPP prototype described here requires FORTRAN to be the base language, but there is no inherent reason why some other imperative language could not be used instead. EPP encourages a structured and readable style of writing parallel programs, and it allows virtually any type of parallelism to be expressed. It maintains as strict a separation as possible between the two main components of a parallel program: semantic actions and logic sequencing. This document is intended for the first-time user of EPP.

1 Introduction

The purpose of this report is to describe the Explicit Parallel Programming (EPP) language and illustrate its use. The intended audience is prospective users of EPP. A companion report [Gamb 91] describes the system in greater detail from an implementation point of view. The remainder of the report is organized as follows. Section 2 precisely defines the syntax of the various components of the EPP language. Extensive examples are given in Section 3, including both code fragments illustrating the various constructs, and complete programs. Finally, in Section 4 we briefly describe the operation of the EPP compiler and executive, and indicate how to invoke the compiler and execute an EPP program.

The broad goal of the EPP project is to investigate several important issues in programming languages for parallel computing. (See [Gamb 90] for a complete discussion.) Perhaps the most important issue deals with the difference between *semantic actions* (the work that needs to be done at run time to solve a problem) and *logic sequencing* (the order in which things get done). A key goal of EPP is to separate these two distinct functions as much as possible. Thus, the EPP language is used to express logic sequencing, and a base language (e.g., FORTRAN) is used for semantic actions. Additional issues addressed in the development of EPP include the importance of being able to use any and all types of parallelism in a program (e.g., not just loops or parallel procedures), and the importance of expressibility and readability in parallel programs. The initial focus of EPP is on shared memory machines; the system described in this report assumes such an environment.

*This work was supported in part by Department of Energy grant DE-FG05-88ER25068.
†First author's current address: IBM FSD, 9500 Godwin Drive, Manassas, VA, 22110.

2 EPP language

This section contains a formal description of the EPP language. The first section formally describes language elements including the recognized character set, a list of keywords, and a description of all other lexical units. The following section describes compilation units—the constant, the type declaration, the data area, and the task. The task is given closer scrutiny in Sections 2.3 and 2.4, covering the syntax and semantics of the component parts, and ways to combine basic tasks into composite tasks. Finally, constructs for iteration and alternation are defined in Section 2.5.

2.1 Language elements

The objects described in this section are those which are recognized and manipulated by the EPP lexical analyzer.

Input to EPP is an ASCII text file. There are actually two different sets of accepted input characters, depending upon whether the compiler is parsing EPP code or source code. (The distinction between source code and EPP code will be made later.) In either case, all control characters (hex 00–1F and 7F–FF) are mapped to the space character. While parsing EPP code, the characters

```
A-Z a-z 0-9 _ " ( ) , : = { }
```

are recognized by the compiler. In the implementation at Virginia Tech, the characters `$ - +` are also recognized and used to access compiler options. Descriptions of the compiler options may be found in Section 4.2. All other characters are illegal. While parsing source code, the compiler will accept any character as input.

EPP has several keywords which must not be used as external names nor as the names of data items. In general, EPP is case sensitive, but with respect to keywords, it is case insensitive. The keywords are

assoc	**clone**	**concur**	**constant**
else	**epp**	**eppend**	**fortran**
if	**inc**	**init**	**loop**
notord	**references**	**serial**	**source**
structure	**type**	**while**	**until**

In addition, the following names should not be used in source code: **eppblock, epp0, epp1, epp2, epp3, epp4.**

Base ten integers are recognized by the EPP compiler. The sole usage of integers in EPP is to dimension arrays. Real numbers are not recognized, because they are not needed. This is an interesting result of the separation of logic sequence and semantic actions.

Lexical identifiers in EPP are a sequence of letters, numbers, and the underscore character, and must begin with a letter. There is no practical limit on the length of identifiers. Lexical identifiers have five uses in EPP. They may be used to identify constant items, data areas, or external procedures, or the may be used as the index of a clone, or as the name of a data item.

EPP recognizes strings delimited by double quotes. A double quote may be included in a string by typing two double quotes: "". Strings are used to name external files, and as constants.

EPP recognizes source code as a lexical type. Source code is syntactically correct, compilable code from some programming language. For the initial implementation, the only language accepted is FORTRAN. Because different languages have their own formats, source code is copied exactly as it appears in the EPP file. This means, for example, that the FORTRAN programmer is responsible

for restricting FORTRAN code to columns 7 through 72, and putting statement labels in columns 1 through 5.

EPP also recognizes source code in special syntactic contexts. The **until** loop, for example, is controlled by an expression written in some compilable language.

```
loop ... until (thecows (int1 + rnd (float1)) .EQ. comehome)
```

Source code which appears as part of a syntactic construction is called an expression, although depending upon the context, the source code may be a complete statement in the source language.

Comments in EPP are delimited by curly braces, and may continue over lines. In general, EPP is a free format language; white space is ignored except when parsing source code. To promote compatibility with other programming languages, several features are suspended while parsing source code (either as straight source, or as an expression). The curly braces no longer function as comment delimiters, and line breaks become significant as statement terminators. Within source code, the programmer must comment in the fashion expected by the source language.

2.2 Compilation units

The EPP ideal of making things easy for the programmer means balancing certain features against programming convenience. For the sake of this balance, structure is limited to the boundaries of compilation units. Other features which contribute to the simplicity of EPP are a small number of lexical types, independent compilation units, and simple ways to combine them.

EPP recognizes four types of compilation units: the constant declaration, the type declaration, the data area declaration, and the task. The constant is self-explanatory, and the type declaration follows ideas used in several languages, but the data area is a new idea, and the task has two varieties—the named task and the unnamed task. The four types of compilation units are introduced in this section.

2.2.1 Constants

The programmer can use constants to parameterize his program and to reproduce long or complicated pieces of code without retyping. Both uses promote modularity in the program. A constant is declared with the following syntax.

```
<constant declaration> = constant <identifier> = <value>
<value> = <integer> | <string>
```

The literal value of the constant is substituted for the constant name in the token stream. Substitution continues even while the compiler is parsing source code, so the user should avoid constant names which are the same as program keywords. In EPP, a constant can be of integer type, or string type. The scope of a constant is from the position of its definition in the text until the end of the file. Since constants are defined external to any other compilation unit, they do act like global names. Attempts to redefine a constant as anything but another constant will generally cause an error.

2.2.2 Types

A **type** declaration reserves certain identifiers as type names. After being so typed by the EPP compiler, that token bears the attribute of **usertype** and can be used to declare data items. When

declaring a type, the programmer indicates the size in bytes of the item, and, optionally, the number of bytes in its boundary (4 for full word, 2 for half word, etc.) and a string which will be used by EPP to make the actual declaration for that data item in an output source file created by EPP. The type declaration has the following syntax.

```
<type declaration> = type <identifier> <size info>
<size info> = <size> | <size> <bounding>
<size> = <integer>
<bounding> = ( <bound> )
<bound> = <integer>
```

A size value of zero is special, and is used to indicate an identifier which should be typed, but which should not allocate memory space. In this case, any bounding information which may be present is ignored. Such semantics are useful when declaring the type of a function name.

2.2.3 Data areas

A **data area** is a contiguous piece of memory. The data area is usually divided up into scalar variables and arrays. In EPP, this is referred to as imposing structure upon the data area. The data area may be a separate compilation unit or may belong to a task. A data area declaration has the following syntax.

```
<area declaration> = structure <data declarations> |
       structure <identifier> <data declarations>
<data declarations> = <data declaration> <data declarations> |
       <data declaration>
<data declaration> = <usertype> <varnames> |
       <usertype> <array_dims>
<varnames> = <identifier> | <identifier> , <varnames>
<identifiers> = <identifier> | <identifier> , <identifiers>
<array_dims> = <array_dim> | <array_dim> , <array_dims>
<array_dim> = <identifier> ( <dimensions> )
<dimensions> = <integer> | <dimensions> , <integer>
```

For arrays, the list of dimensions may include integer constant symbols previously defined, or integer constants. There is no limit on the number of subscripts. However, in EPP, the first element of an array has the subscript 1, which may conflict with some languages (C, for example).

Data areas may also be defined external to any other compilation unit, or may be defined as local to a task. If the data area is defined external to a task, it must be named. The identifier associated with that data area refers to the same piece of memory thereafter. Subsequent data areas may impose a new structure upon that data area, but it will always refer to the same logical piece of memory.

2.2.4 Tasks

An **unnamed task** is a subtask and has no name in its definition. Similarly, the unnamed task cannot accept parameters, although it may declare its own data areas, and has access to variables above it in the task hierarchy, according to Pascal conventions for scoping. The unnamed task provides the programmer with a way to program parallelism "in line". A **named task** is a separately

compiled task which must have a name in its definition. The named task may accept the names of data areas as parameters. A program is necessarily a named task. Whether a program or not, a named task is defined as follows.

```
<named task> = <type> <identifier> <plist> :  <task definition> |
       <type> <identifier> :  <task definition>
<type> = fortran | epp
<plist> = ( <parameters> )
<parameters> = <identifier> | <parameters> , <identifier>
<task definition> = <invocation> | <memory specs> <invocation>
<memory specs> = <data or type> | <memory specs> <data or type>
<data or type> = <area declaration> | <references>
```

An unnamed task, which may appear as a task invocation, has the following structure.

```
<unnamed task> = <invocation> | <memory specs> <invocation>
<memory specs> = <data or type> | <memory specs> <data or type>
<data or type> = <area declaration> | <references>
```

An "invocation" is a task defined by any of the following: a composite task, an **if**, an **if-else**, a **loop**, a **clone**, an **until**-task, or source code. Each of these is further defined in the next section.

2.3 The task and its structure

The base task occupies the lowest semantic level within the EPP system, and is defined exclusively by statements of some compilable language. It is a special design feature of EPP that semantic actions can be written in any compilable imperative language, which saves programmers the trouble of learning an entirely new language. The first approximation of the EPP system, however, restricts semantic actions to the FORTRAN language.

In EPP, tasks are generally built from smaller tasks in a hierarchical structure. The scoping of EPP is such that every subtask has access to types, data items and external references declared by the tasks above it in the hierarchy, and to globally declared types. Scoping is limited to the compilation unit; an external procedure does not inherit the scope of the calling procedure.

The EPP programmer needs to distinguish the terms "invocation" and "task". An invocation indicates a run time event which causes the activation of a task. The EPP program has two methods to indicate invocation. In the explicit method, arguments may be passed to the invoked task. The implicit method indicates invocation by encountering the definition of an unnamed task. The implicit quality of the invocation refers to the fact that while explicit invocations are done by reference through a name, the implicit invocation uses the actual task defined in line instead of a name. Therefore, a named task may only be invoked explicitly; an unnamed task may only be invoked implicitly.

A named task is invoked explicitly with a statement composed of the task name, and an optional series of arguments delimited by parentheses. The parameters to an EPP task are exclusively the names of data areas. If a named task does take parameters, the passed data areas become part of the environment of that procedure.

A **data association** occurs when either the name of a data area follows the keyword **assoc** (at the head of a task) or when a new structure is imposed upon the name of a data area (again, at the head of a task). The ability to impose new structures upon a data area is a convenience for

the programmer. It allows him to change the way he accesses the memory within that data area. In larger programs, it also permits a name to be removed from a data area, which avoids compiler troubles if the name is reused.

Every EPP task may have a **references** section, which allows the programmer to tell the EPP compiler what files to search when attempting to resolve external references. In the reference section, delimited by the keywords **references** and **eppend**, the EPP programmer may identify external tasks from other languages. This enables EPP to format the proper calling sequence for procedures written in compiled languages. The format for an external reference is

```
<reference> = <ref type> <identifier> |
        <ref type> <identifier> <library pathname>
<ref type> = fortran | epp
```

The library pathname is optional; if it is included it will be used when the task is compiled to resolve any external references. A user may also supply a library name to the EPP compiler at compile time. This is described in Section 4.2.

2.4 Composite tasks

EPP is a language for describing logic sequences. Static sequences are described by grouping statements into blocks under one of three orderings: **notord**, **serial** or **concur**. The **serial** heading indicates that all task invocations which are subtasks of **serial** should execute sequentially in the order specified by the text. Task invocations which appear as part of a **notord** block may execute at any time during which the **notord** is active, to wit, until all subtasks terminate. The **concur** heading of a block indicates that all task invocations should occur at the same time.

There is a semantic nuance in the definitions of **notord** and **concur**, which is often not clearly recognized in the actual semantics of statements like **cobegin**. In several references ([Dijk 68, Brin 82, Lisk 82]), the semantics describe the activation of subtasks using the word "simultaneous", which implies an action like **concur**. In [vanW 75], the semantics are not clear but could be those of **notord**. Additionally, tasks which are below a certain threshold size (e.g., smaller than a time slice) cannot be guaranteed to be active at the same time, so **concur** might be considered a **notord** task which includes a suggestion to the executive that all subtasks should be run as close to the same time as possible.

The formal syntax of the composite task is shown.

```
<comp task> = <task type> <invocations> eppend
<task type> = notord | serial | concur
<invocations> = <invocation> <invocations> | <invocation>
```

The handling of source code varies depending upon the task in which the code is found. Quite often, source code will be a series of statements from the source language, not just a single statement. Are the statements to be executed as one task, or as individual tasks? There are instances in which one or the other option is desirable. Fortunately, each situation arises commonly in only certain types of tasks. For **notord** and **concur** tasks, each statement is taken as its own task. Thus, a collection of subroutine calls may execute concurrently. For tasks defined by **serial**, **if**, **loop**, or **clone**, the code is treated as a simple task. until?

The syntax for including code in EPP is to introduce the source code with the keyword **source**.

```
<source code> = source <syntactically-correct source code> epp
```

This suspends many EPP functions. The keyword **epp** returns the compiler to its original functionality. The keywords **notord**, **concur**, **serial**, and **eppend** are still recognized within source code, and allow the programmer to group statements in a manner different from the current manner. This might be handy, for example, if during a **notord** series of subroutine calls, one of the calls is preceded by an initialization statement. The two (or more) lines pertaining to that call could be bracketed with **serial** and **eppend** to obtain the desired function. Although this convention may seem awkward at times, it spares the programmer in the most common cases.

2.5 Iteration and alternation

Dynamic sequences in EPP are described by loop structures which permit iteration, and if-else structures which permit alternation. Almost every compiled language has these control structures, and it may seem redundant to duplicate them in EPP. Control structures, however, describe logic sequences (dynamic sequences, to be precise) and EPP could not meet its objective of giving the programmer explicit control of logic sequence without having its own control structures.

The **if** and **if-else** structures are as in most programming languages. The keyword **if** is followed by an expression and a task; **else** may follow optionally and precedes a task. If there is more than one **if** which may take an **else**, the **else** is paired with the most recent **if**. Since the **if** and **if-else** structures define tasks, the multi-choice if, else-if, else-if, etc. structure may be simulated easily. The formal syntax of **if** is seen here.

```
<if task> = if <source-expression> <invocation> |
        if <source-expression> <invocation> else <invocation>
```

There are actually two kinds of loops from the point of view of the logic sequence. The first is the sequential loop which is familiar from standard programming languages. The second is the loop in which the iterations are executed concurrently. The need for a way to specify concurrent execution of loop iterations is so common that some parallel languages are built around such a loop (BLAZE [Mehr 87], for example). EPP distinguishes the two with different names: **loop** indicates a sequential loop in which the next iteration depends upon some condition, such as a loop index being below the upper bound, or on a logical expression as is common with while-loops; **clone** indicates a loop in which the iterations can execute concurrently, and necessarily, for which the total number of iterations is known before any work is done.

Loops in EPP are described with a combination of the keywords **init**, **while**, **inc**, and **loop**. With these four keywords, the EPP programmer can execute loops that behave like the FORTRAN **DO** loop, or the Pascal **while** loop. The keyword **init** is followed by a source expression which is executed once before the loop begins. This statement can initialize a loop counter, for example. The keyword **while** indicates a source expression which is evaluated before every iteration, and whose result must be true in order for the next iteration to execute. Finally, **inc** introduces a source statement which is executed once after every iteration and before testing the continuation condition. Thus, **inc** may be used to increment a loop counter, for example. Note that **init** and its source expression, and **inc** and its source expression are both optional. Thus, the formal syntax of **loop** is seen below.

```
<loop task> = <loop control> loop <invocation>
<loop control> = <init stmt> <while expr> <inc stmt> |
        <init stmt> <while expr> |
        <while expr> <inc stmt> |
        <while expr>
```

The functionality of the Pascal **repeat-until** loop can be done in EPP with the **loop-until** structure. Its syntax is as follows.

```
<until task> = loop <invocation> until ( <until expr> )
```

Semantic differences between the loops described by **loop** and **clone** were noted earlier. The difference is that the executive must be able to determine the number of iterations of a **clone** independent of any actions which occur inside the **clone**; therefore the **clone** must be controlled by an integral sequence (as opposed to a simple logical condition). Accordingly, all three source expressions (**init**, **while**, and **inc**) are required by **clone**. Additionally, **clone** requires an identifier which will be used as the index of the **clone**. This identifier is treated as an identifier whose existence is limited to the scope of the **clone**. This identifier should be the object of the **init**, **while**, and **inc** actions, but the compiler does not enforce this. The user is free to shoot himself in the foot. The identifier of each task generated by the **clone** will have a unique value. The formal syntax of **clone** is as follows.

```
<clone task> = <clone control> <index> clone <invocation>
<clone control> = <init stmt> <while expr> <inc stmt> |
        <init stmt> <while expr> |
        <while expr> <inc stmt> |
        <while expr>
<indexing> = index ( <usertype> <identifier> )
```

3 Examples

This section illustrates the form and function of the EPP language in more detail through the use of actual code examples. The first part of the presentation will follow the order used in Section 2.2—constants, types, data areas, and tasks. The second part is a series of three example EPP programs, implementing algorithms of increasing complexity, from matrix multiplication to a numerical solution of a partial differential equation. For all examples, the logic sequence will appear as it does in the actual program. For purposes of clarity, the more complicated examples will use pseudo-code for semantic actions and data declarations.

3.1 Constants

A constant may be an integer or a string:

```
constant nrows = 50
constant errormsg = "matrix is not full rank"
```

Thereafter, whenever the compiler finds either the token `nrows` or `errormsg`, it substitutes the values given in the constant definition directly into the token stream. An integer constant may appear in the source code anywhere that an integer may legally appear, including as the dimension of an array.

An important point to note is that the substituted value of a constant is treated as a whole unit; unlike the action of the C compiler and macro preprocessor, the lexical analyzer does not scan the value of the constant.

3.2 Types

The type declaration reserves an identifier so that the identifier may later be used to declare data items. Examples of type declarations for some FORTRAN types would be

```
type    integer     4(4)    "integer"
type    real8       4(4)    "real*8"
type    char12      12(1)   "character*12"
type    real_fnc    0       "real"
```

In some cases, there need be no difference in the spelling of a type name, because that name is a valid EPP identifier. In other cases, the programmer must use a new name, because the actual name would cause errors (e.g., "real*8").

A string constant may be used in a type declaration if so desired. While FORTRAN is limited to a finite number of basic type names, this is not true of C or Pascal, among others. Thus, the C programmer might have an EPP program fragment such as

```
constant process_desc =
     "struct process {
          char              *name;
          int               address;
          int               size;
          struct process    *next;
     }"

type process      16(4)  process_desc
type process_fnc  0      process_desc
```

This example shows an instance in which the programmer has a record structure process which is used to declare data items and to identify the returned value of a function. When a string constant represents the declaration text, double quote marks are not used.

3.3 Data areas

The syntax used to declare data areas serves two purposes. First, it reserves memory sufficient for the data items which are declared. Second, it imposes a structure upon that region of memory, in the sense that the compiler assigns specific addresses to those data items. Whether a data area is defined external to any task, or as a part of a task, both functions are significant. When external to a task, such data areas must be named explicitly. If the same name is used in the definition of another data area, the effect is to impose a new structure upon the same area of memory. If the second structure is larger than the first, EPP accommodates the need for additional memory.

As a part of a task, a data area may or may not be named. If the data area is named, EPP attempts to find a data area in scope or a global data area with the same name. If one is found, the semantics impose a new structure on that area of memory. Otherwise, the data area is local to the task. Data areas which may be in scope include named data areas defined in tasks above the current task, and data areas which are passed as parameters.

A special directive in EPP, the **assoc** command, directs the compiler to include a named data area and its structure in the current task. This is equivalent to (but faster than) defining a data area with the same name and reimposing the same structure upon it.

```
structure global_data
    integer i1, i2, i3
    real8   array(nrow, ncols)
eppend

  ⋮

fortran my_procedure (parameter1) :    { This is a procedure.}
assoc global_data
structure parameter1
    real_fnc cond_number
    integer j1, j2
eppend
structure
    integer pivots(nrows), i4
eppend
⋮
{ Procedure statements go here }
⋮
eppend                          { End of the procedure }

structure global_data
    integer vector1(nrows), vector2(nrows)
    integer permutation(nrows)
eppend
```

Figure 1: Examples of data areas.

Examples of these variations are found in Figure 1. In this example, the first data area is a named global area. It is associated with `my_procedure` through the use of an **assoc** statement[1]. The next data area is a structure imposed upon a data area which will be passed to the procedure. The compiler does not allocate memory in this case, as the memory will already be allocated in the calling procedure. Note that because this data area is named, it may be passed to another procedure. A unnamed local data area follows. A fresh page of memory is allocated and deallocated upon entry and exit, respectively, from `my_procedure`, but it always has the same structure. Any tasks which are subtasks of `my_procedure` will have full access to the data items declared in this data area, or data area `parameter1`. After `my_procedure` the programmer has imposed a new structure upon the global data area. From this point in the file, until the end or the imposition of another structure, the new structure replaces the previous structure, although the same area of memory is referenced.

The syntax of declaring data items is straightforward. In EPP, the declaration of data items requires a type name, followed by a list of one or more items. Commas separate the items in the list, and every item in the list is given the type indicated. Examples of data item declarations appear in the examples of data areas.

3.4 Tasks

In this section, we give examples of the basic composite tasks **notord**, **serial**, and **concur**, and the dynamic sequence tasks **if**, **if-else**, **loop**, **until**, and **clone**. In the examples, the reader will often see "invocation" representing the execution of some task. Be reminded that an invocation may be an external source language procedure, an unnamed EPP task, or a reference to a named EPP task. Also, the examples are indented to show program structure. This need not be, since the language is free format—spacing has no significance in EPP. Finally, EPP comments will appear delimited by curly braces.

Examples of the basic composite tasks—**notord**, **serial**, and **concur**—appear in Figure 2. These three keywords each pair with **eppend** to create a task from several smaller invocations. The semantics are that within a **notord** task, the invocations may occur in any order. Within a **serial** task, the invocations occur in the order given by the text. The **concur** task requires all invocations to be activated at the same time, although nothing implicitly coordinates their execution beyond that point. Any of the EPP tasks described in this section may appear as an invocation in any other task.

Examples of **if** and **if-else** are given in Figure 3. The **if** behaves as experience with other languages would lead one to believe. If the condition in an **if** is true, the following invocation is executed. If false, the invocation is skipped. Remember that the code which appears as the condition is source code—FORTRAN, in this case.

Examples of the dynamic sequence tasks generated by **loop**, **until**, and **clone** are given in Figure 4. The **loop** consists of three keyword identified source code expressions or statements which determine the execution characteristics of the **loop**. The keyword **init** identifies a statement which is executed once before the **loop**. **While** identifies an expression which must be true before the execution of each **loop** iteration. Once the condition tests false, the **loop** terminates. **Inc** identifies a statement which is executed after each iteration of the **loop** and before the continuation condition is tested. Again, all code appearing in parentheses is FORTRAN code. Only the **while**

[1]The choice of **assoc** as a keyword may be misleading, since a data association only maps memory into a program and has nothing to do with structure. In spite of this distinction, **assoc** still seemed like the most descriptive term for the semantics.

```
notord
    invocation_1
    invocation_2
    ⋮
    invocation_n
eppend

serial
    invocation_1
    invocation_2
    ⋮
    invocation_n
eppend

concur
    invocation_1
    invocation_2
    ⋮
    invocation_n
eppend
```

Figure 2: Composite tasks.

```
if ( I_were_a .gt.  rich_man )
    dibbie_dibbie_dibbie ()     { invocation }

if ( I_had .eq.  a_hammer )
    Id_hammer_in_the_morning () { invocation }
else
    Id_hammer_in_the_evening () { invocation }
```

Figure 3: Examples of **if** and **if-else**.

```
init (myid = 1) while (myid .lt.  maxid) inc (myid = myid + 1) loop
    my_favorite_routine()       { invocation }

while ( myid .lt.  maxid ) loop
    my_favorite_routine()       { invocation }

loop
    bop()                       { invocation }
until ( you .lt.  drop )

init (myid = 1) while (myid .lt.  maxid) inc (myid = myid + 1)
index (integer myid) clone
    send_in_the_clones ( array ( 1, myid ))
```

Figure 4: Examples of **loop**, **until**, and **clone**.

control is mandatory. Omission of **inc**, in particular, allows the programmer to configure a **loop** which acts like a Pascal **while** loop. Regardless of which controls are present, they must be in the order shown.

The **until** behaves exactly like the while loop described above with two exceptions. First, the continuation condition, which is identified by the keyword **until**, is logically inverted. That is, the loop continues to execute iterations while the condition evaluates false. Second, the semantics are such that **until** always executes one iteration.

The **clone** generates any number of copies of an invocation, each of which has a special **index**. The copies of this invocation execute as if they were part of a **notord** task. **Clone** uses the same controls as **loop**, but all controls must be present. Additionally, **clone** has the **index** parameter, which is generally used to subscript arrays being passed to a procedure. The index variable need not be declared anywhere in the program—its appearance in the clone serves as its declaration, which is made so that every clone gets its own local copy of the variable.

3.5 Matrix multiplication program

The first full program presented is a matrix multiplication program (see Figure 5). The algorithm is rather simple, assuming that there exists a few helpful library functions. There is plenty of parallelism in this exercise, and it shows several nice features of EPP. The program assumes that there are subroutines on the local directory which may be found in files `fill.o` and `mat.o`. The function of `fill_array` is to put random floating-point values in an array having the given dimensions; `mat_col_mult` is a subroutine which does matrix-vector multiplication and returns the result in a vector.

At the highest level, the program fills two arrays with random numbers, multiplies the two arrays, putting the result in a third array, and then prints out the product array. These tasks are all grouped under a **serial** heading.

At the next level of refinement, the task which fills two arrays with random numbers invokes two instances of `fill_array`. These invocations may proceed in any order, so they are grouped under a **notord** heading. The next task clones several matrix-vector multiplication tasks. Each of

```
constant dim1 = 20
constant dim2 = 50
constant dim3 = 40


typeinteger  4    "integer"
typereal     4    "real"


fortran matrix_mult:            { The program starts here }


structure
    real    a(dim1,dim2)
    real    b(dim2,dim3)
    real    c(dim1,dim3)
    integer ix1
    integer ix2
eppend
references
    fortran fill_array    "fill.o"
    fortran mat_col_mult  "mat.o"
eppend
{-----------------------------------------------------------------}

serial
{ Put data in arrays a, and b }
    notord
        fill_array(a,dim1,dim2)
        fill_array(b,dim2,dim3)
    eppend
{ The actual multiplication can proceed in parallel }
    init (myid = 1) while (myid .le.  dim3) inc (myid = myid + 1)
    index (integer myid) clone
        mat_col_mult(a, b (1, myid), dim1, dim2, c(1, myid))
{ Print out the product matrix c }
    source
        do 120 ix1 = 1, dim1
            write(*,1001) (c(ix1,ix2),ix2 = 1,dim3)
1001        format(10(2x,i5))
120     continue
eppend
```

Figure 5: Matrix multiplication program.

these may run simultaneously, because each task has a distinct portion of the data to compute.

Note that while writing the program, the programmer does know how many `mat_col_mult` tasks there will be, since he wrote that number as the value of `dim3`. However, it is much preferable to write one **clone** than a **notord** task with 40 individual invocations. With the **clone**, the programmer can change the value of `dim3` and still have a correct program. This is not possible using **notord**. In contrast, the clone would not work to call both `fill_array` invocations since the arguments use two different matrices.

The source code which prints the results is customized for the array by virtue of the constants. Their appearances as controls of the implied `do` loops will be replaced at compile time by the appropriate numbers.

3.6 Matrix inversion program

The next program is a more complicated example which performs matrix inversion (see Figure 6). The program is the next increment of complexity and shows extensive use of source code throughout the program. Because of its size, pseudo-code is used to represent large chunks of code which have little illustrative value.

The top level of the program is a sequence which first generates a random matrix, using a FORTRAN subroutine `rndmat`. For this example, we assume that all external references are stored in a library which is given as a command-line parameter. Therefore, no explicit filenames are needed in the **references**. Next, the program makes a copy of the new array to be used later when checking the result. Then an LU factorization of the matrix is computed. All work to this point has been done in source code. The programmer is not obligated to call FORTRAN routines from EPP; this feature is merely a convenience. In the case shown here, the programmer deemed it more efficient to collect all his source code into the largest task possible.

The program then generates the inverse by solving $AA^{-1} = I$ for the columns of A^{-1}. Finally, the program checks the accuracy[2] of the solution is by computing $||AA^{-1} - I||_\infty$.

3.7 Schwarz splitting program

Schwarz splitting is a parallel method for solving elliptic PDEs over a decomposed two-dimensional domain. The domain is split into overlapping subdomains, each of which is solved independently. The overlap between subdomains is the means by which the individual subdomains communicate with each other. The algorithm has fairly large granularity, which is the reason for its popularity as a method for solving such problems.

Once again, the full program is too large to include in the text, so many details are omitted. The value of this example is as a demonstration of structured description of an algorithm—smaller tasks used to create larger tasks. The example in Figure 7 has three levels of tasks.

The program solves a PDE over a grid with the following steps. After initializing grids, and creating a matrix system to approximate the unknown function, it factors the matrix system. This is done only once, since the factorization remains unchanged by any subsequent actions. All subdomain problems must be factored before the program enters an **until**, which continues until the problem determines convergence. Within the **until**, the current solution values of all subdomains are saved in an array called `unkold`. This array is used for testing convergence. With the factorization saved earlier, the program then does a forward and back solve on the matrix system of each subdomain to compute a new solution. The program then determines the maximum

[2]Note that this is *not* recommended as a robust method for checking the accuracy of LU factorization.

```
  ⋮
{ Constants, type declarations, and data areas }
  ⋮

fortran inverse :
serial
    source
        errmax = 0.0d0
c generate random matrix, save a copy of it, and factor
        call rndmat (a_new, dim, dim)
        do 10 ix1 = 1, dim
            call dcopy(dim, a_new(1,ix1), 1, a_sav(1,ix1), 1)
10      continue
        call dgefa (a_new, dim, dim, ipvt, info)
    epp
c solve for columns of a_inv
    init (ix2 = 1) while (ix2 .le.  dim) inc (ix2 = ix2 + 1)
    index (integer ix2) clone
        source
            do 90 ix1 = 1, dim
                a_inv(ix1,ix2) = 0.0d0
90          continue
            a_inv(ix2,ix2) = 1.0d0
            call dgesl (a_new, dim, dim, ipvt, a_inv(1,ix2), 0)
        epp
c check that a * a_inv - I is small
    init (ix2 = 1) while (ix2 .le.  dim) inc (ix2 = ix2 + 1)
    index (integer ix2) clone
        source
            do 190 ix1 = 1, dim
                temp = ddot(dim, a_sav(ix1,1), dim, a_inv(1,ix2), 1)
                if (ix1 .eq.  ix2) then
                    err = dabs(temp-1.0d0)
                else
                    err = dabs(temp)
                endif
                errmax = dmax1(errmax, err)
190         continue
        epp
eppend
```

Figure 6: Matrix inversion program.

```
   ⋮
fortran solve:
   ⋮
      { type declarations, data areas here }
   ⋮

serial
      initialize_data_structures_flags_and_counters( )

      init (kpc = 1) while (kpc .le. npcs) inc (kpc = kpc + 1)
      index (integer kpc) clone
            generate_matrix_system_for_grid_piece(kpc)

      init (kpc = 1) while (kpc .le. npcs) inc (kpc = kpc + 1)
      index (integer kpc) clone
            factor_and_save_factorization_for_piece(kpc)

      loop
            serial
                  init (kpc = 1) while (kpc .le. npcs) inc (kpc = kpc + 1)
                  index (integer kpc) clone
                        copy_to_unkold_for_piece ( kpc )

                  init (kpc = 1) while (kpc .le. npcs) inc (kpc = kpc + 1)
                  index (integer kpc) clone
                        forward_and_back_solve_for_piece ( kpc )

                  find_difference_between_new_and_old_solutions ( dfmax )
                  source
                        lsdone = (dfmax .lt. eps) .or. (iter .ge. maxits)
                  epp
            eppend

      until (lsdone)
eppend
```

Figure 7: Schwarz splitting program.

difference between the new and old solutions. This value is used to determine convergence. The loop terminates when either the solution has converged, or the **until** exceeds some predetermined number of iterations.

4 Compilation and execution

4.1 Compiler actions

The EPP compiler reads a single program containing EPP code. Compilation consists of parsing the input file for correct syntax, identifying source code and source expressions, and invoking the proper compiler to produce executable code from those, and generating a number of output files which are logically linked. The central output file is the *logic sequence file.*

In general, the output from most parallel compilers for shared memory machines is an image file containing a single task image. At places indicated by the program, the original image spawns child processes which are under control of the original process. For the initial implementation of EPP, this is not the case. The EPP compiler produces several output files: one which describes the logic sequence to the EPP executive, and one file for each simple task. A simple task is a text file generated by the EPP compiler which includes some source code written by the user. These files are normally sent to a compiler for compilation, but there is a compiler option which preserves the text files. Directions for running the compiler appear in Section 4.2.

The novelty of task execution in the EPP system is that there may be several independent tasks running at the same time. In this instance, independent means that the tasks are activated independently of each other. There is no parent-child relationship among them. These independent tasks are coordinated by an executive task (the EPP executive). At execution time, the EPP executive will read the logic sequence file, determine which tasks will run first, supply the correct data associations, and initiate the tasks. A description of the logic sequence file is found in [Gamb 91]. As tasks return completion status, the EPP executive initiates more tasks for execution. The program is done when the final task returns its completion status.

If the logic sequence is completely determined by a static sequence, the only tasks which need to run are those which perform semantic actions. Whenever there is a dynamic sequence, an additional task must run to evaluate the condition which determines the logic sequence from that point. Typically, these tasks will be small logical expressions. Handling such small tasks efficiently is one of the challenges of implementing EPP.

The EPP executive is an additional piece of baggage in the EPP system in that it consumes machine cycles without directly contributing to the execution of the program at hand. There are two good justifications for the existence of the executive, however. First, and foremost, the EPP system design is to make parallel programming easier for humans, not computers. Other design considerations in the EPP system compelled the decision to have multiple, independent tasks, which made necessary an overseeing executive task. Second, an executive can be tuned to dynamically improve performance, somewhat offsetting the penalty for its existence. Load balancing algorithms can be adapted to particular machines, resulting in EPP programs whose execution adapts to the host computer without any changes to the program itself.

4.2 Compiler invocation and options

The EPP compiler is invoked with the command

```
eppcmp fname { library }.
```

The file named by `fname` is read and processed; `library` is the optional name of a library file. If present, the name of that file is included in the command which compiles and links source code text files. This mechanism provides a way for the programmer to provide several external references without having to identify the location of each.

The compiler assumes the existence of a dedicated directory named "epp" under the current directory. Into this directory, the compiler writes all the image files which it may need during execution. If the directory is not present, the result will be a run-time error in the compiler. The central logic sequence file is stored in the current directory under the name "AREA0".

The compiler offers a set of options which can be helpful during program development. The options listed below are turned on and off using the following format in the source code.

```
{ $ option-symbol plus/minus
```

The left curly brace appears literally, indicating the beginning of a comment (which must eventually be closed by a right curly brace). The dollar-sign also appears literally, and indicates that compiler options follow. The option-symbol is a single letter which identifies a particular compiler option. The plus/minus is either a plus sign or a minus sign. A plus sign turns the option "on", while a minus sign acts to the contrary. These four symbols must appear without intervening white space. Additional options may appear separated from preceding options by a comma, as in {`$L+,T+,...`}.

Recognized compiler option-symbols are

- `L` List EPP source code.
- `T` List EPP tokens.
- `P` List SLR compiler production steps, as they occur.
- `S` List EPP symbol table for each task.
- `H` List the EPP task structure for each named task.
- `C` Suppress compilation of source code—output is text files as they appear before compilation.

4.3 Executing an EPP program

To run an EPP program, the user types

```
epprun name {-cm}
```

to activate the EPP executive and execute his program. In the command sequence, `name` is the name of the central logic sequence file—"AREA0", unless changed by the user. The options act as follows:

`c` Prints the command which invokes individual processes as they are initiated by the executive. The command consists of a file name, an offset into the central logic sequence file, and an integer which indicates the number of environments for this process.

`m` Monitor execution by printing a hierarchical representation of the process run-time tree after every change to the tree. Actions which cause a change to the tree include the queueing of a task, the activation of a task, and the completion of a task.

The user should be warned of one pitfall during execution. The executive depends upon each task reporting its completion status upon termination. This action is done automatically by EPP

tasks which exit gracefully. Should a task abort for any reason, the executive will continue to wait for a message of completion which will never come. At this point, the user must abort the executive, correct the error, and try again.

References

[Brin 82] Per Brinch Hansen, "Edison—A multiprocessor language", *Software Practice and Experience*, 11(1982), pp. 325–361.

[Dijk 68] Edsger W. Dijkstra, "Cooperating sequential processes", in *Programming Languages*, F. Genuys, ed., Academic Press, New York, 1968, pp. 43–110.

[Gamb 90] Jim Gamble, *Explicit Parallel Programming*, MS Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1990.

[Gamb 91] Jim Gamble and Calvin J. Ribbens, "Explicit Parallel Programming: system description", Research Report TR 91-20, Department of Computer Science, Virginia Polytechnic Institute and State University, July, 1991.

[Lisk 82] B. L. Liskor and R. Scheifler, "Guardians and actions: linguistic support for robust distributed programs", in *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1982, pp. 7–19.

[Mehr 87] Piyush Mehrotra and John van Rosendale, "The BLAZE language: a parallel language for scientific programming", *Par. Comput.*, 5(1987), pp. 339–361.

[vanW 75] A. van Winjgaarden, et. al., "Revised report on the algorithm language ALGOL68", *Acta Informatica*, 5(1975), pp. 1–236.