EXTENSIONS TO THE C PROGRAMMING LANGUAGE
FOR SIMD/MIMD PARALLELISM

James T. Kuehn
Howard Jay Siegel

PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907 USA

Abstract
A superset of the C programming language that is applicable to the SIMD/MIMD mode processing environment of PASM is described. The language extensions for SIMD mode include the definition of parallel variables, functions, and expressions; a scheme for accessing parallel variables; and extended control structure semantics. Extensions for MIMD mode are realized by defining a preprocessor to convert a generalized CSP-like language to standard C with operating system calls inserted to support the parallelism. Extensions to the libraries of I/O and operating systems functions used in parallel mode are also discussed. The PASM parallel processing system is used as an example target machine.

I. Introduction
There are currently two general strategies for programming parallel processors. One is the definition of languages that allow the programmer to express the parallelism of a problem explicitly. Parallel languages that have been proposed/implemented thus far have either been targeted toward SIMD machines where a given number of processors is assumed at compile time (e.g., Glynis [10], Actus [14, 16], Lucas [4], Vector C [11]) or toward multiprocess systems using the Concurrent Sequential Process (CSP) model (e.g., Concurrent C [13], Occam [5], Ada tasking capabilities [2]). Another strategy is the adoption of existing serial languages for parallel machines, relying on extensive analysis during compilation to extract the parallelism (e.g., KAP and VAST [2], Parafras [7]). This paper employs the first strategy by defining extensions to the C programming language [6] for the SIMD and MIMD modes of parallelism [8].

PASM is a partitionable SIMD/MIMD system [17] which can be structured as one or more independent SIMD and/or MIMD machines. PASM consists of N Processing Elements (PEs), Q Microcontrollers (MCs), a partitionable multistage interconnection network, multiple secondary storage units, and additional processors dedicated to job scheduling and I/O. There are N/Q PEs associated with each MC. A virtual SIMD or MIMD machine of MN/Q PEs is formed by combining the efforts of M MCs. The MCs act as control units for their PEs in SIMD mode and perform scheduling and memory management processors is being constructed at Purdue University using Motorola MC68010 processors and other off-the-shelf components [12].

Some of parallel-C's goals and philosophies are outlined in Section II. In Section III, new constructs are described for SIMD mode. Section IV discusses support for MIMD mode.

II. Goals and Philosophies
In keeping with the tradition of C, the language extensions proposed attempt to minimize the number of new keywords, constructs, and idioms. Extensions that can reasonably be implemented as a call to a system- or user-supplied function are preferred over new operators that would make the compiler more complex. The C programming language has found favor with systems and applications programmers alike because it is highly efficiently compiled, has modern control structures, and allows the user to get very close to the machine hardware if desired. The laissez-faire tradition of C is also continued; execution continues unless a program error is non-recoverable; i.e., accesses to nonexistent or protected memory, communication with PEs not assigned to the task, etc.

Processors are a resource just as is memory. In serial languages, the variable declarations indicate how much memory is to be allocated to the program. If additional memory is needed during execution, a system function can be called to provide it. In practice however, the amount of memory desired may outstrip the physical memory available. Virtual memory techniques are used to overcome this difficulty. Extending this concept to parallelism introduces the notion of a Virtual PE (VPE). If a parallel program has a declaration of an array containing N elements, it is natural therefore, to have the program indicate the extent of the parallelism (number of VPEs desired) and for the compiler to map the needs onto the existing hardware. Techniques for performing this mapping have been described for other languages (e.g., [11, 14, 15]). Parallel-C provides a way to specify the initial allocation of VPEs desired at compile time and allows system functions to be called to allocate/deallocate VPEs as necessary at run time.

III. Extensions for SIMD Mode
Declarations of parallel variables and functions
Similar to other parallel languages, both scalar (normal) and parallel variables may be defined. These types of variables are introduced with two new keywords: scalar (the default) and parallel. When parallel variables are introduced, they are usually defined in their extent. The parallel keyword is followed by one or more subscripts that define the shape of the parallelism. Assuming N and MAXLINE are compile-time constant expressions, some declarations of parallel variables are:

```
  parallel [N] int a;
  parallel [N] char line[MAXLINE];
  struct node{
    char word;
    struct node *next;
  } parallel [N] nodespac[100], *head;
```

which define for each of N VPEs one integer 'a,' an array of MAXLINE chars, an array of 100 nodes, and a node pointer. The shape of the parallelism may also be multi-dimensional as in:

```
  parallel [2][2][2] int a;
```

which declares 'a' to be an integer in each of 8 VPEs configured as a 2-by-2-by-2 array. Multi-dimensional parallelism often aids the formulation and coding of certain multi-dimensional problems (e.g., image processing).

Consider the three declarations:

```
  parallel [100] struct node pnode;
  scalar struct node cnode[100];
  parallel [N] struct node pnode[100];
```

The first two declarations allocate exactly 100 variables of type struct node. The 100 "nodes" are distributed one to a VPE.
while the 100 "cnodes" are allocated in the control unit (or to each PASM MC participating in the virtual SIMD machine). The last declaration allocates 1,000N nodes in total, 100 in each of N VPEs.

The type of variable a function returns is known as the "type of a function." Extending this concept to parallel functions, suppose that the square root of a parallel variable is to be calculated. Each VPE will calculate the square root of one element of the parallel variable. The function will return a double precision floating point (double) value in each of the VPEs; therefore, its type declaration in the calling function will be:

```
parallel [N] double psqrt();
```

and the definition of the parallel function itself is:

```
parallel [N] double psqrt(x)
```

```
parallel [N] double answer;
+ calculation of square root +
return(answer[0]);
``` 

Note that the function declaration and definition have empty brackets. This is because functions can be defined to work for any extent of parallelism; the actual extent is determined at run time based on the extents of the function's arguments and returned value. The notation "answer[*]" selects all of the current extent and is discussed in the next section.

**Selecting parallel variables**

When operating on parallel variables, references along the parallel dimension(s) can be done at the same time and will be referred to as selectors. Selection is typically done using special variables called selectors (like the selectors in [4]). A selector is a boolean bit vector of any shape that is used to define the set of VPEs that will perform a certain operation. For some target machines, the selector keyword allows the compiler to generate special instructions so that hardware dedicated to enabling and disabling VPEs may be used. In others, selectors are compiled as a special type of parallel variable. For example:

```
selector [N] mask1;
sselector [2][4] mask2;
```

define mask1 to be a selector for an array of N VPEs and mask2 to be a selector for a 2-by-4 array of VPEs. Arrays of selectors, pointers to selectors, etc. may also be defined.

The initialization of selectors can be a tedious process since the extent of the parallelism can be large. Therefore, a simplified notation based on PE Address Masks [17] is allowed. For N VPEs, a log_2N -position mask (a PE address mask) specifies which of the N VPEs are to be selected. Each position of the mask corresponds to a bit position in the logical numbering of the VPEs and consists of a 0, 1, or X (don't care) specification. VPEs whose addresses match the mask 0 matches 0, 1 matches 1, and 0 or 1 match X) are selected. Mask specifications are grouped with braces, expressions indicate repetition factors, and square brackets denote a complete mask specification. For example:

```
[N] [n-i{0}i{X}] enables VPEs 0 to 2^{N-1}.
```

Examples of selector initialization are:

```
selector [N] odds = [n-i{0}i{X}];
sselector [4] evens = [1, 0, 1, 0];
```

The first example expands the PE address mask into the aggregate initializer [0, 1, 0, 1, 0, ...] and assigns it to the selector. The second example shows how the aggregate initializer can be specified directly (without a mask). Selectors may be complemented, "or"-ed, "and"-ed, "xor"-ed, or differenced to obtain the complement, union, intersection, equivalence, or set difference. A special mask, [""], selects all VPEs. In Actus and Vector-C, the "start/(increment)/finish" construct (or its equivalent) and the set operators are used to build "index sets." This is an equivalent method for expressing the extent of parallelism, but the resulting index sets are static for these languages. Selectors are dynamic and if specified using masks, more space-efficient.

When a parallel variable of a certain shape is referenced, the selector(s) used must match the shape. For example,

```
parallel [N] int a; /* declaration */
a[odds]; /* reference */
```

selects the value of 'a' in the odd-numbered VPEs. Similarly,

```
parallel [N] int a[10]; /* declaration */
a[odds][0] /* reference */
```

selects the value of a[0] in the odd-numbered VPEs. "0" is the index of the first element of the array 'a' within a VPE.

When scalar variables or constants are used to select parallel variables, only one element of the parallel variable is accessed. Parallel variables may be used to select and index other parallel variables. For example,

```
parallel [N] int a[10], b, c; /* declarations */
ab[0][c] /* reference */
```

causes 'a' to be selected in VPEs where the local element of 'b' is non-zero. Indexing of array 'a' within a VPE is determined by the local value of 'c'.

```
```

To perform the operation in all VPEs, but to select the returned results from only even-numbered VPEs, the following would be used:

```
parallel [N] double y, psqrt(); /* declarations */
psqrt(y[0][0]); /* function call */
```

Finally, to perform the operation in only the even-numbered VPEs and to select only the even-numbered VPEs results:

```
psqrt(y[0][0][0][0]);
```

is used. Note that from the point of view of the caller, the last two forms return the same result. However, some functions that are passed different ranges of parallelism, i.e., [""] vs. [evens], may produce different side effects due to inter-PE data transfers, using the extent of parallelism in expressions, etc.

All active VPEs evaluate their arguments when a function is called. Some VPEs might be temporarily inactive due to conditional statements as described later.) Associated with each actual parallel parameter in the parameter list, there is a selector that indicates which elements of the variable are to be used in expressions. A copy of the value of the parameter and its associated selector is placed on the stack of all active VPEs. In the first two cases above, y's associated selector is [""]; therefore, each active VPE stacks the value of 'y' followed by the corresponding element of the selector (each VPE stacks a 1). In the third case, y's selector is [evens]; therefore, all active VPEs stack 'y' but then even-numbered VPEs stack 1s and odd-numbered VPEs stack 0s.

For each formal parameter of a given type declared in the called function there must be a corresponding actual parameter of the same type in the call. Assume that within the psqrt() function, fpy is the name of the formal parameter. While every VPE active at the time of the call to psqrt() "executes" the function, accesses to 'fpy' are restricted to those VPEs for which 'fpy' is selected (VPEs having the corresponding selector '1' on the stack). For example, if the actual parameter 'y' had selector [evens] as in "psqrt([evens])", but inside the function, 'fpy' was used in an expression as "fpy[0]", the value of 'fpy' would be selected only for the even-numbered VPEs. If inside the function, 'fpy' was used in an expression like "fpy[0][0]", where 'first' selected VPEs numbered 0-3, the value of 'fpy' would be defined only for VPEs 0 and 2. This is because within the scope of the called function, 'fpy' was defined only for even-numbered VPEs; the 'first' selector further narrows the selection set within the function. This mechanism allows selection of expressions within a function.
The conventions for returning values are similar. An expression returned by a function may be defined for all VPEs, e.g., "return(answer[])," but only those results in the VPEs selected by the calling function are used.

**Expressions and assignment**

Scalar and parallel variables in expressions and assignments can be combined as long as no parallel shape conflicts occur. Expressions containing parallel variables always result in a parallel value. The rules are similar to those employed in other SIMD languages; they are:

1. Left side scalar; right side scalar. This is the normal C assignment statement.
2. Left side scalar; right side parallel. The right side should have one and only one element selected. Selecting other than one element is not an error, but the results are undefined.
3. Left side parallel; right side scalar. The right side is promoted to the parallel type and shape of the left side. Selected elements of the left side are assigned the value of the scalar.
4. Left side parallel; right side parallel. The selected elements of the left side are assigned the corresponding values of the right side expression. The range of the selected items on the right side should equal or overlap that of the left for the results to be defined.

In an assignment where both the right and left sides are parallel, corresponding elements are assigned. For example, parallel[N] int a, b; /* declarations */

\[a[0] = b[0]; /* reference */

does not move the value of 'b' from VPE 3 to 'a' in VPE 2. In fact VPE 2's 'a' becomes undefined since no element of the right hand side is selected for VPE 2. Special functions (described later) are used to move data values from VPE to VPE.

**Control structure**

C has five different constructs for affecting control flow: if, then-else, switch-case, while, for, and do. The first two are used to select different execution paths, while the remaining constructs control repetition of statements. The semantics of each of these constructs have been extended for parallel mode in ways identical to earlier SIMD languages (e.g., Argo, LUCAS). For example, in the parallel equivalent of an "if-then-else" construct:

```c
if ( [parallel-expression] )
  [then-block]
else
  [else-block]
```

the [parallel-expression] can be thought to define a selector which is "true-valued" for some VPEs and "false-valued" for others. Thus some of the VPEs will perform [then-block] while others will execute [else-block].

In a strict SIMD environment, [then-block] and [else-block] cannot be executed by the VPEs simultaneously since there is a single instruction stream. Thus, side effects from [then-block] may affect [else-block]. PASM, which can operate in SIMD or MIMD mode, can temporarily switch to MIMD mode so that [then-block] could be executed in parallel with [else-block]. Any references to parallel variables within the block are narrowed to the extent of parallelism within the block. For example, if the [then-block] is executed only by even-numbered VPEs, and a parallel variable within the block is referenced with [+] only the even-numbered elements of the variable are selected.

In SIMD mode, PEs receive their instructions from the control unit and process data in their local memories. In MIMD mode, PEs have both data and program in their local memories. For PASM, whenever a PE accesses a certain instruction address range AR, it operates in SIMD mode and an instruction request is issued to the control unit. When all active PEs associated with the control unit have requested, the control unit broadcasts an instruction to them. The control unit switches the PEs from SIMD to MIMD mode by broadcasting an unconditional jump to the beginning of the MIMD program to the PEs, where the address of the MIMD program is outside AR. The PEs (independently) return to SIMD mode by jumping into the address range AR. When all PEs have returned, SIMD processing continues. As an alternative, the control unit could broadcast a "jump subroutine" instruction to begin MIMD mode and the PEs could "return" to SIMD mode.

This is one example of the effect of allowing both modes of parallelism in a single system. Optimizations such as this are also possible for the switch-case and the looping constructs.

**Functions for data alignment and I/O**

There are no "standard" or "built-in" functions in C; libraries of functions written in C and assembly language are provided at load time. Users may define or redefine these functions by calling the underlying operating system primitives directly. In this section, a small subset of operating system primitives and user-level functions for PASM will be described.

**System Calls.** Functions for allowing a PASM PE to obtain its virtual and physical address are necessary because each PE generates its own interconnection network routing requests. For example, if PE i wants to communicate with PE j + 2, it must first obtain its i, add 2, and write the resulting address to the routing register associated with its port to the network.

- `parallel(if int getvirprocnf, getphyprocnf);`
- `are the functions that can be used to obtain the information.`

The network used in PASM [10] is set for an allowable one-to-one or broadcast (one-to-many) permutation when PEs establish connections by writing the desired destination addresses and broadcast tags to their network routing registers. The function used is `icnbset(destaddr, broadcastmask)` where both destaddr and broadcastmask are parallel unsigned integers.

The transfer of a parallel integer variable "data" using an established setting of the network is performed by `icntransfer(data)` which returns a parallel integer.

Blocks of data are transferred using an established setting of the network by `icntransfer(srcptr, destptr, nbytes)` where 'srcptr' and 'destptr' are parallel character pointers indicating the source and destination addresses of the block to be transferred and 'nbytes' is a scalar integer indicating the block size.

**User-level Calls.** User-level functions provide a higher-level interface than do the system calls given above. For example, `cubef(data, i, N)` performs a cube, interconnection function (transfer data between N PEs whose addresses differ only in the ith bit position [16]) using parallel integer variable "data," returning a parallel integer. This function calls `icnbset()` and `icntransfer()` to carry out the transfer.

Similarly, `shufflewrite(srcptr, destptr, nbytes, N)` sets up the shuffle interconnection function [18] among N PEs and transfers blocks of "nbytes" bytes among them. Because the PASM network cannot perform the shuffle function in one pass, the shufflewrite is implemented with multiple calls to `icnbset()` and `icnwrite()`.

There are dozens of other functions at this level for data alignment (e.g., shift and rotate), for testing (e.g., obtaining the "first" non-zero element of a selector), determine "if all" VPEs are active, and for parallel I/O and mathematical functions.

**IV. Extensions for MIMD Mode**

The constructs given in the last section allowed a single program (instruction stream) to access and manipulate data items located in multiple VPEs. MIMD mode implies that each PE will execute its own instructions from its own memory; thus none of the SIMD constructs are necessary in MIMD mode: MIMD programs will compile using the serial C compiler. However, MIMD processes must coordinate by communicating with each other and by synchronizing the order of their execution. Furthermore, processes operating on shared data structures must do so in a mutually exclusive manner to maintain data integrity.
We propose to use the approach suggested by ConCurrent C [13]: to define a C meta-language with new constructs and keywords that provides general mechanisms for process interaction, control, concurrent execution, event supervision, and sharing. An operating-system-specific preprocessor is used to convert the extended language to serial C with operating system calls to support the parallelism. Therefore, the MIMD extensions would not affect the C compiler. Of course, MIMD programs may call functions to start up SIMD processes and vice-versa, so the eventual goal is a preprocessor that accepts a set of serial/SIMD/MIMD programs, converts the MIMD portion to serial C (with system calls), and passes the serial/SIMD portions through unchanged. The extended (SIMD) compiler would then be called to compile the resulting serial/SIMD program.

V. Summary

A superset of the C programming language that is applicable to the SIMD/MIMD mode processing environment of PASM has been described. The language extensions proposed for SIMD mode are, in general, similar in concept to those introduced in other SIMD mode languages. Some of the new ideas here include the dynamically-settable selectors that indicate the current extent of parallelism, the use of PE address masks, and the handling of the "if-then-else" construct in SIMD mode. The extensions for MIMD mode are much more system-related than language-related. This is a desirable attribute since the development of the compiler for the SIMD extensions and the translator for the MIMD extensions can proceed independently.

We propose to develop a compiler for the language using the PASM prototype as a target. The target assembly language and object file formats for PASM prototype serial, SIMD, and MIMD programs have already been developed and have been in use for some time [9]. The first steps toward the compiler extensions for SIMD mode, modifying the lexical and syntactic analysis routines, are being made. The set of operating system calls and user-level functions required to support the parallelism in both SIMD and MIMD modes is also being developed and refined. Work on the code generation phases of the compiler will proceed in stages.

Initially, the compiler will assume a static allocation of physical PEs (i.e., PVE = physical PE). Next, dynamic allocation of VPEs up to the maximum physical machine size will be allowed. The first "production" compiler will allow dynamic allocation of VPEs to any number, but will assume a fixed-size physical machine is available. Therefore, code compiled for a physical machine of 64 PEs will run only when 64 PEs are available. An eventual research goal is the development of a machine-size-independent program representation that allows the operating system to choose the physical machine size on which to run the program. This is desirable in the PASM environment where the sizes of the available virtual machines vary.

References