ABC: A Blocked C/C++ Parallel Programming Model

Mike Holly and Bill Homer, Cray Research, Inc.,
655-F Lone Oak Drive, Eagan, Minnesota 55121

ABSTRACT: A model is proposed for programming massively parallel machines in the C and C++ languages. This model, which addresses both data and execution parallelism, is designed to achieve that parallelism with only small user-program modifications. The model is initially proposed for implementation on massively parallel (MPP) machines, but should also be useful for parallel-vector (PVP) and symmetric multiprocessing (SMP) architectures.

1 Introduction

The current Cray C/C++ parallel-vector (PVP) machine environment provides parallel operations via microtasking (where parallelism is explicitly stated with user directives) and autotasking (where parallelism is implicitly applied to loops via an automatic optimizer). In either of these PVP models, execution begins with a single thread, and breaks into multiple parallel threads in well-defined regions and tasked loops. A thread is not associated with any particular processor.

On Cray Research’s massively parallel (MPP) machines (Cray-T3D™, Cray-T3E™), the current parallel C and C++ approach has three components:

• message passing via calls to PVM library routines
• fast get/put operations on distributed data via shmem library calls
• a few intrinsics (_num_pes and _my_pe)

The current MPP model has multiple execution threads from the beginning of the program, with each thread corresponding exactly to one physical processor (PE).

In this paper we propose a new paradigm for parallel C/C++: the “ABC” (A Blocked C/C++) parallel programming model. This model is intended for eventual implementation on both PVP and MPP machines, thus unifying parallel programming across a diverse set of hardware. It is designed to minimize the changes needed to convert an existing serial program into a parallel program.

2 Goals for a Parallel C/C++ Model

The ABC parallel C/C++ model has been developed to meet a number of goals. Among the more important are:

• Require only modest changes to normal C/C++ serial programs.
• Provide high speed execution.
• Be implementable and supportable using current technology.
• Provide support for distributed data.
• Provide prior art for future standards (unfortunately, no present standards exist in this area).
• Be compatible with PVM (quasi-standard message passing).

We believe that the ABC model will meet all of these goals.

3 Framework

A complete programming model for C/C++ needs to address two major areas: data distribution and work distribution. Data distribution is particularly important for MPP machines, where there is a significant performance penalty associated with accessing non-local data; this is much less of a problem on PVP machines where all of memory is directly accessible to all processors. For work distribution, work should be done in parallel as often as possible, and the distribution of the work should be such as to minimize the elapsed time. On MPP machines, this often means that work should be done where the data resides, to minimize communication costs (“owner computes” rule).

In the ABC model we assume, first of all, that execution begins in parallel, i.e. that there are some number n of threads which are active at the start of the program. On an MPP machine this is assumed to correspond to the number of physical processors assigned to the program, which may be chosen by the user at compile, link, or load time. We also assume that, by default, all data is replicated on each processor, i.e. all data is by
default private. Within this framework, the ABC model provides for both data distribution and work distribution.

4 Data Distribution

The main question in data distribution is how to partition a data array across the local memories of the \( n \) processors. Many partitions are possible. Some of the more common distributions are

- cyclic: the array is considered linearly, and each successive element is assigned to the next processor in sequence, wrapping around to processor 0 after processor \( n-1 \);
- blocked: some rectangular subset of the array is assigned to each processor;
- degenerate: the entire array, or some dimension(s) of the array, is assigned to a single processor.

In the ABC model we choose to allow a combination of blocked and degenerate distribution, requiring that at least one dimension of a distributed array be designated as “blocked” by using a new keyword block. The general syntax for distributed array declarations is:

\[
type\ array\_name[\text{dim 0}][\ldots][\text{dim} m\ block][\ldots][\text{dim} n];
\]

where the keyword block may appear in any or all dimensions as the last item in the brackets. The meaning of such a declaration is that the array will have its blocked dimension(s) spread as evenly as possible across the available processors. For an array with a single blocked dimension, this implies a block size of:

\[
\text{ceiling}(\text{dimension} / \text{number of processors}).
\]

As an example, consider the array declaration

\[
\text{int } a[20][50 \text{ block}];
\]

In this declaration the leftmost dimension is “degenerate”, i.e. it appears in full on all processors. If compiled (or linked or loaded) for 4 processors, the subarray \( a[0-19][0-12] \) would reside on processor 0, the subarray \( a[0-19][13-25] \) on processor 1, etc. The resulting arrangement is shown graphically in Figure 1.

When more than one dimension is blocked, we intend that the compiler will allocate the array in a fashion which tends to minimize the so-called “surface-to-volume” ratio, i.e. to make each \( n \)-dimensional subblock as “round” as possible. The exact algorithm for doing this is not yet specified. But as an example, consider the declaration

\[
\text{int } a[20 \text{ block}][50 \text{ block}];
\]

If compiled for 64 processors, the compiler algorithm could consider the processor grid as a 4x16 array, and choose to put something like a 5x4 subarray on each of the first 8 processors, and a 5x3 subarray on the remaining 56 processors. Intrinsic functions, described later, will be provided to allow the user to determine the exact layout chosen by the compiler.

4.1 Distributed Arrays

The inclusion of the keyword block designates a new C/C++ derived type, the “distributed” array. Such an array will obey the current C/C++ rules for arrays, except that

- the memory for the distributed array is allocated in local memory across the available processors;
- no pointer object can contain the address of a subobject of a distributed array (no declaration involving the distributed array may omit the blocked dimension or any dimensions to the right of the blocked dimension);
- distributed arrays are not allowed as members of structs or unions;
- there are implications for loops operating on distributed arrays, as described later.

Some of these restrictions are illustrated in the code examples in Figure 2 below.

4.2 Local Operations on Distributed Arrays

ABC also provides two operators for doing local operations on distributed arrays, local_address and local_sizeof.

The local_address operator is useful for accessing the locally resident portion of a distributed array via a pointer. For the array declaration

\[
\text{int } a[20][50 \text{ block}];
\]

we allow writing

\[
\text{int } * \text{ local\_a } = \text{local\_address}(a);
\]

which causes local_a on each processor to point to the beginning of the portion of \( a \) which is stored in the local memory of that processor. Figure 1 shows how the values of local_a are set for this example.

An additional operator, local_sizeof, evaluates to the size of that portion of a distributed array that is resident on a given processor:

\[
\text{local\_sizeof(array, proc\_number)}
\]

provides the size of the portion of \( array \) which resides on processor \( proc\_number \). The array parameter may be either the actual name of a distributed array, or simply a distributed array type.
Using the above declarations, the array \( a \) can be set to zero using the local pointers:

\[
\text{for } (i = 0; i < \text{local_sizeof}(a, _MY_PE)/\text{sizeof} (\text{int}); i++) \\
\text{ \quad } \*\text{local}_a++ = 0;
\]

The `local_sizeof` operator is also useful in allocating a dynamic distributed array:

\[
\text{int } (*q)[20][50 \text{ block}];
\]

\[
q = \text{shmalloc(local_sizeof(*q, 0));}
\]

Here processor 0 is specified, because its local share of the array will have maximal size. The `shmalloc` function allocates memory from the shared heap; there is a corresponding `shfree` function which must be used to release memory allocated via `shmalloc`.

### 5 Work Distribution

There are many possibilities for data distribution on a parallel system, and even more possibilities for work distribution. We consider that the most important case involves the C `for` loop, and the distribution of the work in such a loop to the various processors.

In the ABC model, the directive

\[
\#pragma _CRI shared_on <expr>
\]

is provided to designate a loop or loop nest for shared execution. The iterations of any loop which immediately follows this directive are parcelled out among the available processors, each processor executing a portion of the loop in parallel with all of the other processors. The expression `<expr>` is used to determine which processor executes a particular iteration of the loop: it is exactly that processor where the lvalue expression `<expr>` resides. This expression must designate a distributed array element, using indices which have the general form \( a*i+b \) where \( a \) and \( b \) are constants and \( i \) is a loop control variable for the shared loop. None of the loop control variables may be used in the expression more than once.

There is also an implicit `barrier` at the end of such a loop. That is, all processors will be held at the loop termination point until all other processors have finished with the loop execution.

The `shared_on` directive may not be nested; users are responsible to ensure, for example, that a function called from within a shared loop does not itself contain a `shared_on` directive. Undefined behavior results if this restriction is violated.

### 6 Examples

With this much of the ABC programming model specified, we are in a position to provide a few examples.

#### 6.1 Matrix Multiply

An example for matrix multiply is shown in Figure 3.

Here the columns of the three arrays will be distributed as evenly as possible across the available processors. The loop nest will have a portion of its iterations done on each processor; each processor will execute, for all values of \( k \), those parts of the loop for which \( a[i][j] \) resides on the processor. The loop control variables in the expression must be those for the tight loop nest following the directive; that is, in this example \( i \) and \( j \) are permitted, but not \( k \).

```c
main() {
    float a[100][100 block], b[100][100 block], c[100][100 block];
    int i, j, k;
    \#pragma _CRI shared_on a[i][j]
    for (i=0; i<100; i++) {
        for (j=0; j<100; j++) {
            a[i][j] = 0.0;
            for (k=0; k<100; k++) {
                a[i][j] += b[i][k]*c[k][j];
            }
        }
    }
}
```

Figure 3: Matrix Multiply Example

---

**Figure 2: Code Examples**

Using the above declarations, the array \( a \) can be set to zero using the local pointers:

\[
\text{for } (i = 0; i < \text{local_sizeof}(a, _MY_PE)/\text{sizeof} (\text{int}); i++) \\
\text{ \quad } *\text{local}_a++ = 0;
\]

The `local_sizeof` operator is also useful in allocating a dynamic distributed array:

\[
\text{int } (*q)[20][50 \text{ block}];
\]

\[
q = \text{shmalloc(local_sizeof(*q, 0));}
\]

Here processor 0 is specified, because its local share of the array will have maximal size. The `shmalloc` function allocates memory from the shared heap; there is a corresponding `shfree` function which must be used to release memory allocated via `shmalloc`.
Note that the code and the result are the same as would be achieved in a normal serial matrix multiply, except for the block keyword and the shared_on directive.

6.2 Matrix Transpose

In the next example, given in Figure 4, we show how a matrix transpose function might be written using the ABC model. We assume that the underlying compiler already provides a variable-length-array (VLA) syntax (as do Cray’s and GNU’s C compilers). The VLA syntax is an integral part of the ABC proposal.

Here, with 4 processors and \( m = n = 100 \), there would be 10000 accesses of \( b \) on the left side of the assignment, and 10000 accesses of \( a \) on the right. All of the reads of \( a \) will be local, since the shared_on expression is \( a[j][i] \), but there will be 7500 remote stores to \( b \).

Note that:
a) the remote/local accesses in these cases are probably optimal, or at least a better arrangement is not obvious; and
b) the code will have the same result in a non-parallel environment, with no change required except to define away the block keyword and ignore the pragma directive.

```c
void mtranspose(int m, int n, float b[m][n block],
                  float a[n][m block]) {
    int i,j;
    #pragma _CRI shared_on a[j][i]
    for (i=0; i<m; i++) {
        for (j=0; j<n; j++) {
            b[i][j] = a[j][i];
        }
    }
}
```

Figure 4: Matrix Transpose Example

7 Built-in Macros

The ABC model provides several built-in macros for frequently needed functionality. These are:

- \_N_PES which resolves to the number of threads (number of processors in the MPP case) for which the program was compiled (or linked or loaded).
- \_MY_PE which is the index of the currently executing thread, a number between 0 and (\_N_PES - 1)
- \_HOME_PE(variable) which designates the thread (processor) where variable resides; variable must be a shared scalar or a fully dereferenced distributed array element.

8 Intrinsic Functions

While the essential ABC model presented so far can be used to easily parallelize many sequential C programs, more is needed. There are also two intrinsic functions which are a part of ABC.

- \_in_master() returns TRUE if execution is in a master region, FALSE otherwise
- \_blksize(arr, dim, proc) provides a count of the number of elements of the distributed array arr in dimension dim which reside on the designated processor proc; the dimension is numbered from the leftmost dimension which is dimension 1.

9 Additional Directives

The basic work and data distribution algorithm, which spreads the distributed array and the work to be done evenly across the available processors, is expected to be useful in most cases. However, for those instances where more control is needed, ABC provides the following set of directives:

- #pragma _CRI master
- #pragma _CRI endmaster[,copy(var[ var,... var])] delimit a serial region. executed only by processor 0, with all other threads waiting at the start and end of the master region. The variables in the copy list must denote private scalar or array objects, which are copied from processor 0 to all other processors when processor 0 reaches the endmaster directive. Objects with type bit-field or char are not allowed in the copy list.
- #pragma _CRI critical
- #pragma _CRI endcritical define a critical region; only one processor enters at a time, without regard to order; no synchronization occurs.
- #pragma _CRI atomic_update enforces serial access to the single statement which follows the directive; this is useful to prevent potential race conditions.
- #pragma _CRI shared var[,var,... var] designates shared scalar variables or arrays, i.e. variables and arrays which will reside in the memory of only one processor (processor 0). Each var must be a simple variable name; see Figure 5 for some examples.
- #pragma _CRI symmetric var[,var,... var] designates local variables which will be placed at the same local memory offset on each processor.
- #pragma _CRI barrier
- #pragma _CRI nobarrier respectively create an explicit barrier and eliminate an implicit barrier.
10 A Further Example

Figure 6 gives an additional example which illustrates the use of some of the ABC macros and directives. This example is derived from a similar program in reference [2]. It estimates a value for $\pi$ by counting the number of randomly-generated (x,y) coordinate pairs which fall within a unit circle. The trials are run independently on all processors, and then the results are summed in a reduction which is executed by each processor, one at a time, in indeterminate order.

In this example the first ABC construct is the shared directive at line 17, which causes the variable totalhits to exist only on processor 0. Neither the block keyword nor the shared_on directive are needed for this very parallel example.

At line 18 the total number of trials is broken up into trials-per-processor (my_trials). At lines 21-22 the major part of the work is done. Since this loop is not controlled by any directives, each processor will execute the entire loop.

At line 24, the atomic_update directive ensures that only one processor at a time will update the totalhits value.

Finally, at line 27, the directive establishing a master region ensures that the calculation of $\pi$ and the printf are executed only on processor 0. Since there is an implied barrier at this directive, this calculation does not begin until all processors are finished with their loop calculations and have made their contribution to the totalhits sum.

11 Summary

ABC is a model for C and C++ parallel programming. It is designed to make use of parallelism simple, and only requires a single keyword to designate data arrays which are to be processed in parallel, plus a single directive to designate parallel loops.

12 References


