

Extending Parallel Pseudo-Code Language Peril-L

Peiyi Tang

Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Abstract

In this paper, we propose to extend the Peril-L parallel pseudo-code language. The extensions added to the Peril-L are essential to express non-trivial parallel algorithms. We demonstrate the effectiveness of the extended Peril-L by showing the parallel algorithms for the Jacobi method for solving differential equations and the dynamic programming method for finding the optimal binary search tree.

1 Introduction

To write parallel programs for parallel computers, people need to design parallel algorithms or parallelize sequential algorithms before they can implement them in parallel programming languages. For traditional sequential computing, algorithm designers have been using the well-established pseudo-code or pidgin-code languages [1] to describe rigorous algorithms without details of the implementation in programming languages. The algorithms in pseudo-code can also be used to analyze and predict the performance accurately, because all the pseudo-code languages are based on the accurate RAM (Random Access Machine) model for all sequential computers.

The situation is quite different in the parallel computing world. While many parallel programming languages (Fortran 95, HP Fortran, OpenMP and MPI libraries, Pthread library, etc.) have been developed, there was virtually no parallel pseudo-code language proposed or developed until recently. Most parallel algorithms for scalable parallel computers, have been described by using the combination of diagrams, illustrations and texts as demonstrated by many parallel algorithms and programming texts such as [2]. There were parallel algorithms expressed in the pseudo-code based on the PRAM (Parallel Random Access Machine) machine model. However, these algorithms are either impractical (due to the assumption of unlimited number of processors) or inaccurate in performance prediction (due to the assumption of shared memory and uniform memory access in the PRAM model).

The practical and scalable parallel machines such as IBM BlueGene [3] and SiCortex SC5832 [4] do not have shared memory. They all can be modeled by the CTA (Candidate Type Architecture) model, where each processor consists of one or several CPUs, the memory and the network interface. All processors are connected by an interconnection network. All memories are local to its CPUs and there is no global shared memory.

To the best of our knowledge, the first pseudo-code language to describe scalable parallel algorithms for CTA machines is Peril-L¹ proposed by Lin and Snyder recently [5]. Peril-L can describe rigorous scalable parallel algorithms without dealing with the low-level details of partitioning and communication and yet it can expose the cost of communication for accurate performance analysis.

The major components of the Peril-L pseudo-code language are parallel threads, global variables and their partitioning, and full/empty synchronization variables. Partitioning of a global virtual array, specified by three functions: `mySize()`, `localize()` and `localToGlobal()`, is to map the disjoint portions of the array to different threads. This is the key to allowing parallel algorithm designers to specify the SPMD (Single Program and Multiple Data) parallel algorithms without details of the partitioning.

In recent efforts to write parallel algorithms with Peril-L, we found that in order to express non-trivial parallel algorithms we need to extend it with notations for (1) parallel thread grids, (2) partition description vectors, (3) accessing size information and variables of non-local threads, and (4) linking local array variables. In this paper, we present these extensions and justify them by showing the parallel algorithms for the Jacobi method and the dynamic programming method in the extended Peril-L. Section 2 introduces the original Peril-L. The extensions to the Peril-L along with the parallel Jacobi algorithm are presented in Section 3. Section 4 presents the parallel algorithm of the

¹According to Lin and Snyder, the name was chosen to be homophonic with "parallel", not to imply that there is something dangerous about it.

dynamic programming method for the optimal binary search tree in the extended Peril-L. Section 5 concludes the paper.

2 Peril-L Notation

Peril-L extends the basic programming facilities in C and provides notations for (1) parallel threads, (2) partitioning of virtual global memory and connection between global and local memory, and (3) synchronization through the full/empty synchronization memory. The global variables in the global (virtual) memory are declared on the top level and their names are distinguished from local variable names by being underlined. The parallel threads are declared as the top-level `forall` block as follows [5]:

```
int allData[n];
forall (threadID in (0..P-1))
{
    int size = mySize(allData[]);
    int locData[size] = localize(allData[]);
    ...
    int g = localToGlobal(allData[], j, i);
    ...
}
```

All variables declared within the `forall` block are local variables of the thread. The partitioning of global arrays among the parallel threads are defined by three functions: (see above code)

- `mySize(allData[])` returns the size of the partition of global array `allData[]` for the thread. It is assumed that the sizes of the partitions for different threads differ by no more than 1.
- `localize(allData[])` returns the local 0-origin reference of the partition of the global array. Therefore, `locData[]` as the return value of `localize(allData[])` as above is not a local variable, but rather a local reference to the partition of the global array for the thread. Reading or writing of `locData[size]` is done to the corresponding partition of the global array.
- `localToGlobal(allData[], j, i)` returns the global index corresponding to the local index `j` of the `i`-th dimension of the partition.

Peril-L allows for full/empty (FE) synchronization variables in the global virtual memory which are distinguished by an `'` at the end of the variable name. Peril-L provides the barrier synchronization primitive `barrier` for barrier synchronization across all threads.

A critical section can be specified by an exclusive block expressed as `exclusive{...}`.

Peril-L also provides the reduction and scan functions for associative and commutative operators like `+`, `*`, `min`, and `max`. The reduction using `+` for the sum of array `a[]` is expressed as `sum = +/a[]`. The scan (prefix) using `+` for the partial sums before each element of `a[]` is expressed as `prefix[] = +\a[]`. The reduction for the sum of scalar local variable `locv` in all threads can be expressed as `sum = +/locv`. The reduction and scan operators imply a barrier synchronization across all threads. The details of the Peril-L notations can be found in Chapter 4 of [5].

3 Extending Peril-L

To make the Peril-L pseudo-code language suitable to express general scalable parallel algorithms, we extend it with the notations for (1) parallel thread grids, (2) partition description vectors, (3) reading size information and variables of non-local threads, and (4) linking local array variables.

Parallel Thread Grids

The partitioning of global arrays is closely related to the structure of parallel threads. The parallel threads should be allowed to be structured and specified as a multi-dimensional grid. We extended the `forall()` notation of Peril-L to allow a multi-dimensional thread grid to be defined in one line. The syntax of the extended `forall()` is `forall(tiv in irv){ tbody }`, where `tiv` is the thread index vector, `irv` the index range vector, and `tbody` the body of thread code. A two-dimensional thread grid with `t1`×`t2` threads can be specified as:

```
forall ((p1,p2) in (0..t1-1,0..t2-1))
{
    ...
}
```

where `(p1,p2)` is the 0-origin thread index vector. With this `forall()` notation, the rank² 2 of the thread grid is explicitly specified by the thread index vector `(p1,p2)`. The understanding of the rank of the thread grid is essential to specifying the partitioning of global arrays through `mySize()`, `localize()`, and `localToGlobal()` functions.

Sub-Array and Partition Description Vector

Some parallel algorithms may require parallel threads to work on different parts of the global arrays at different stages. Therefore, it is important to be able to

²The rank of a thread grid or a data array is the number of dimensions of the grid or array.

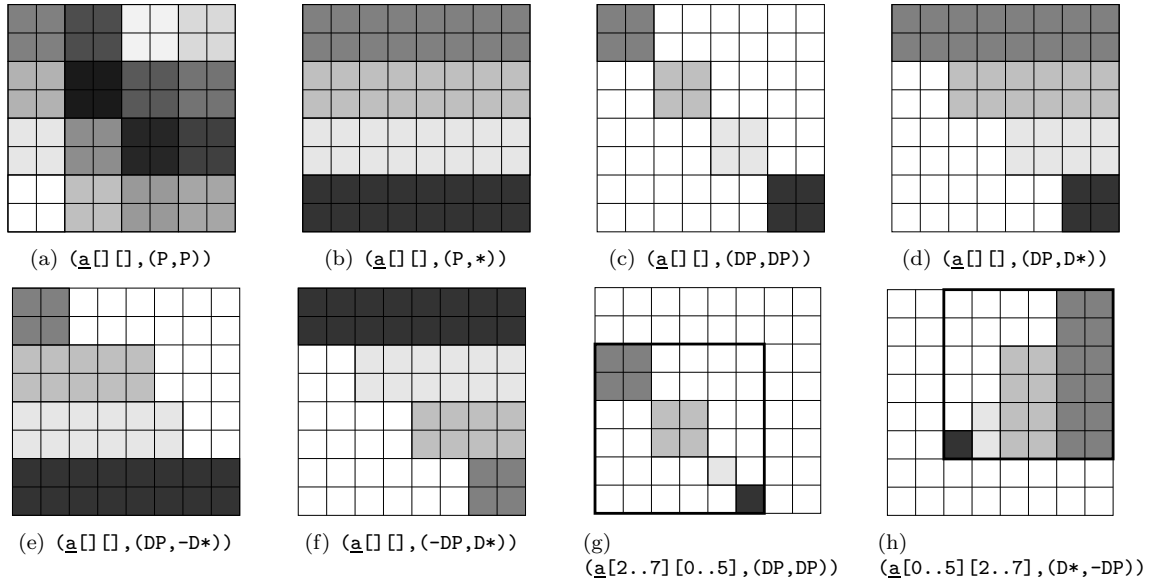


Figure 1: Partitioning of Global Virtual Arrays

specify the partitioning of the sub-arrays of global arrays. Sub-arrays of an array $\underline{a}[n][m]$ are specified as $\underline{a}[n1..n2][m1..m2]$ where $0 \leq n1 \leq n2 \leq n-1$ and $0 \leq m1 \leq m2 \leq m-1$. The first parameter of functions `mySize()`, `localize()` and `localToGlobal()` now can be the whole array specified as $\underline{a}[\][\]$ (or simply \underline{a}) or a sub-array $\underline{a}[n1..n2][m1..m2]$.

In order to specify a variety of partitions of arrays or sub-arrays, we use a so-called *partition description vector* as the second parameter of functions `mySize()`, `localize()` and `localToGlobal()`. For an n -dimensional array or sub-array, a partition description vector is an n -vector with each of its elements either equal to P , $-P$, DP , $-DP$, D^* , $-D^*$, or $*$. P means block partitioning and $*$ means flooding in the corresponding dimension. Therefore, $(\underline{a}[\][\], (P,P))$ means the 2-dimensional (2D) partitioning of a 2D array \underline{a} on a 2D thread grid. The extended Peril-L pseudo-code for this partition would be as follows:

```
float a[8][8];
forall ((p1,p2) in (0..3,0..3))
{
  int s1,s2;
  (s1,s2) = mySize(a[\ ][\ ], (P,P));
  loca[s1][s2] = localize(a[\ ][\ ], (P,P));
  ...
}
```

This partitioning is shown in Figure 1(a). $(\underline{a}[\][\], (P,*))$ represents the row-strip partitioning of the 2D array \underline{a} on a 1D thread grid as shown in Figure 1(b). DP and D^* represent the diagonal partitioning and diagonal flooding, respectively. Thus, $(\underline{a}[\][\], (DP,DP))$ and $(\underline{a}[\][\], (DP,D^*))$ represent the di-

agonal partitioning of 2D array \underline{a} in both dimensions as shown in Figure 1(c) and the diagonal partitioning with flooding at the second dimension as shown in Figure 1(d), respectively. The $-$ sign in front of P , DP , D^* indicates the reverse direction of the partitioning or flooding. For instance, $(\underline{a}[\][\], (DP, -D^*))$ specifies the diagonal partitioning with reverse flooding in the second dimension shown in Figure 1(e). Likewise, $(\underline{a}[\][\], (-DP,D^*))$ specifies the reverse diagonal partitioning with flooding shown in Figure 1(f). Notice that the order of the shades representing different threads is reversed in the first dimension.

The combination of the sub-array notation and the partition description vector has the power to specify almost any partitioning. The sub-array partitionings by the same 1D thread grid of 4 threads shown in Figure 1(g) and Figure 1(h) can be specified by $(\underline{a}[2..7][2..7], (DP,DP))$ and $(\underline{a}[0..5][2..7], (D^*,-DP))$, respectively.

Accessing Local Arrays and Size Information of Others Threads

To allow to express data communication between threads directly, we extended Peril-L to allow the reading of local variables of other threads. Given a local array variable, say `work[\][\]`, for every thread, a thread can read part of `work[\][\]` in another thread `tid` by putting `work<tid>[n1..n2][m1..m2]` in the right-hand side of an array assignment statement. Reading local variables of other threads needs to be properly synchronized and this can be done by using `barrier` synchronization or FE synchronization variables in Peril-L.

The sizes of the partitions of a global array for different threads may be different. Many local arrays are dynamically allocated in threads according to their different sizes. In order to read these local arrays precisely, a thread needs to know their sizes in other threads. We extended the Peril-L's `Size()` function to get the size information of other threads. The syntax of the extended `Size()` function is `Size<tid>(g,pdv)`, where `tid` is the thread identity vector of the thread from which to get the size information, `g` the global array or sub-array, and `pdv` the partition description vector. Therefore, the Peril-L function `mySize()` is a special form of `Size<>()` function with the thread identify vector of the current thread.

In addition to the extensions described above, we also extend Peril-L to include array assignments. We also allow non-zero origin arrays for the convenience of algorithm writing. A non-zero origin array can be declared using index ranges starting from non-zeros such as: `float work[-2..n+1][3..m]`.

Figure 2 shows the parallel pseudo-code in the extended Peril-L for the Jacobi method for solving differential equations. The sequential algorithm of the Jacobi method is as follows:

```
float a[n+2][n+2], b[n+2][n+2];
initialize a[] [];
int err = h+1;
while (err > h) {
    err = 0;
    for (i=1; i<=n; i++) {
        for (j=1; j<=n; j++) {
            b[i][j] = (a[i-1][j]+a[i+1][j]+
                a[i][j-1]+a[i][j+1])/4;
            err = max(err, |b[i][j]-a[i][j]|);
        }
    }
    a[1..n][1..n] = b[1..n][1..n];
}
```

In the parallel pseudo-code in Figure 2, the sub-array `a[1..n][1..n]` of the global array `a[n+2][n+2]` is partitioned by the 2-dimensional `t1×t2` thread grid. Each thread gets the size of its partition in the size vector `(s1,s2)`. Local array `work[s1+2][s2+2]` is the working array for the local partition of `a[] []` and it includes the four ghost areas of width 1, `work[1..s1][s2+1]`, `work[1..s1][0]`, `work[s1+1][1..s2]`, and `work[0][1..s2]`, in the East, West, South, and North edges, respectively (`work[0][0]` is the North-West corner of the array). The code in the beginning of the `while` loop is to update the four ghost areas by the corresponding elements of the `work[] []` array of the neighbor threads. Note that for updating the ghost areas at the West and North edges, each thread needs the size information of its West and North neighbor

threads. For instance, the size information of the partition of the West neighbor is obtained by calling `Size<p1,p2-1>(a[1..n][1..n],(P,P))` and stored in `(ws1,ws2)`. Then, the West ghost area `work[1..s1][0]` is updated by the values read from its West neighbor, `work<p1,p2-1>[1..s1][ws2]`. The reason is that the `s2` of its West neighbor (stored in `ws2`) may be different from its own `s2`. The last statement `err = max/err` of the `while` loop is the max reduction of all `errs` in all the threads. Note that pseudo-code in Figure 2 does not have message passing which should be part of implementation details. The algorithm is concise and rigorous. It is scalable and works for any size of thread grid even 1×1 grid with one thread.

```
float a[n+2][n+2];
forall ((p1,p2) in (0..t1-1,0..t2-1)) {
    (s1,s2) = mySize(a[1..n][1..n],(P,P));
    loca[s1][s2] = localize(a[1..n][1..n],(P,P));
    float work[s1+2][s2+2];
    float b[1..s1][1..s2];
    work[1..s1][1..s2] = loca[0..s1-1][0..s2-1];
    initialize boundaries of work[] [] from a[] [];
    barrier;
    err = h+1;
    while (err > h) {
        if (p2+1 <= t2-1)
            work[1..s1][s2+1] = work<p1,p2+1>[1..s1][1];
        if (p2-1 >= 0) {
            (ws1,ws2)=Size<p1,p2-1>(a[1..n][1..n],(P,P));
            work[1..s1][0] = work<p1,p2-1>[1..s1][ws2];
        }
        if (p1+1 <= t1-1)
            work[s1+1][1..s2] = work<p1+1,p2>[1][1..s2];
        if (p1-1 >= 0) {
            (ns1,ns2)=Size<p1-1,p2>(a[1..n][1..n],(P,P));
            work[0][1..s2] = work<p1-1,p2>[ns1][1..s2];
        }
        barrier;
        err = 0;
        for (int i=1; i<=s1; i++) {
            for (int j=1; j<=s2; j++) {
                b[i][j] =(work[i-1][j]+work[i-1][j] +
                    work[i][j-1]+work[i][j+1])/4;
                err = max(err, |b[i][j]-a[i][j]|);
            }
        }
        work[1..s1][1..s2] = b[1..s1][1..s2];
        barrier;
        err = max/err;
    }
    loca[0..s1-1][0..s2-1]= work[1..s1][1..s2];
}
```

Figure 2: Parallel Pseudo-Code of Jacobi Method

Linking Local Arrays

The computation of a thread often needs to refer to the data read from other threads and stored in buffers. To express the algorithm of the local computation, it is often necessary to rename these buffers in line with the major local data structure. In the extended Peril-L, we propose to use link statement to link extensions of the major data structure to those buffers. The syntax of the link statement is `link ref to buff`, where `ref` is the link array and `buff` the local buffer array which `ref` is linked to. The link array `ref` should have the same shape and size with the buffer array `buff`. Link array `ref` is just a reference to the buffer array `buff`; there is no local copy of `buff` in `ref`. We will see an example of the link statement in the next example of parallel pseudo-code in the next section.

4 Parallel Dynamic Programming

In this section, we present the parallel pseudo-code in the extended Peril-L, for the optimal binary search tree problem, to further demonstrate the expressive power and effectiveness of the language.

Given n keys, $K_1 < \dots < K_n$, and their probabilities of occurrence, p_1, \dots, p_n such that $\sum_{i=1}^n p_i = 1$, the problem is to find the binary search tree (BST) for those keys with the minimum average search time.

The optimal binary search tree problem can be solved by using dynamic programming method. If the optimal binary search tree for K_1, \dots, K_n has K_r ($1 \leq r \leq n$) as its root, then its left sub-tree containing K_1, \dots, K_{r-1} and right sub-tree containing K_{r+1}, \dots, K_n must both be optimal. Let the optimal BST containing $j - i + 1$ keys, K_i, \dots, K_j ($j \geq i - 1$), be denoted by $BST_{i,j}$ and its mean search time (MST) by $MST_{i,j}$. Note that $BST_{i,i-1}$ is an empty tree and, thus, $MST_{i,i-1} = 0$. $MST_{i,j}$ can be calculated by the following recursive equation:

$$MST_{i,j} = \min_{i \leq r \leq j} (MST_{i,r-1} + MST_{r+1,j}) + \sum_{k=i}^j p_k \quad (1)$$

The value of r that gives the minimum of the sums $MST_{i,r-1} + MST_{r+1,j}$ determines K_r as the root of $BST_{i,j}$.

The working data structure for finding $MST(1, n)$ is an $(n + 1) \times (n + 1)$ upper-triangular matrix `cost[1..n+1][0..n]`. The matrix element `cost[i][j]` is used to store $MST_{i,j}$. Note that the index ranges of the first and second dimensions of `cost` are $[1..n+1]$ and $[0..n]$, respectively. `cost[k][k]` ($1 \leq k \leq n$) and `cost[k][k-1]` ($1 \leq k \leq n + 1$) are p_k and 0, respectively.

The dynamic programming algorithm for finding the optimal binary search tree is as follows [6]:

```
float cost[1..n+1][0..n]; int root[1..n+1][0..n];
float prob[1..n];
main() {
    for (i=n+1; i>=1; i--)
        for (j=i-1; j<=n; j++)
            MST(cost, prob, root, i, j);
}

MST(cost, prob, root, i, j) {
    if (j < i) {
        cost[i][j] = 0;
        root[i][j] = -1;
    } elseif (i==j) {
        cost[i][j] = prob[i];
        root[i][j] = i;
    } else {
        psum = +/prob[i..j];
        bestCost = MAX;
        bestRoot = -1;
        for (int r=i; r<=j; r++) {
            rCost = psum + cost[i][r-1] + cost[r+1][j];
            if (rCost < bestCost) {
                bestCost = rCost;
                bestRoot = r;
            }
        }
        cost[i][j] = bestCost;
        root[i][j] = bestRoot;
    }
}
```

To compute the `cost[i][j]` for $MST_{i,j}$, we only need the elements of the matrix to its left (up to the diagonal), `cost[i][i-1..j-1]`, and the elements below (up to the diagonal), `cost[i+1..j+1][j]`. The computations of those `cost[i][j]` that $j-i$ is constant can be executed in parallel. In other words, the parallelism lies along the diagonals of the matrix and is one-dimensional. Therefore, the thread grid is one-dimensional with \underline{t} threads in total. Without loss of generality, we also assume that $n+1$, one plus the size of the problem n , is a multiple of \underline{t} . If $n+1$ is not a multiple of \underline{t} , we can always add more keys with probability of occurrence 0 to make $n+1$ a multiple of \underline{t} . Adding these zero-probability keys will not change the problem and $BST_{1,n}$ is still the optimal binary search tree of the original problem.

We use the diagonal partition with flooding (DP,D*) to partition the `Cost[][]` and `Root[][]` arrays as shown in Figure 1(d). The probability vector `Prob[]` is duplicated in every thread, because it is one-dimensional and read-only.

The parallel algorithm in the extended Peril-L pseudo-code language is shown in Figure 3. Given the \underline{t} threads, there are \underline{t} diagonals. The parallel execution of a diagonal is controlled by the `for` loop of index `k` (see Figure 3). Also, the number of threads required to execute each diagonal varies. The threads that join

```

float Cost[1..n+1][0..n]; int Root[1..n+1][0..n];
float Prob[1..n];
forall ((p) in (0..t-1)) {
    int t = t; int n = n;
    (s1,s2) = mySize(Cost[ ][ ], (DP,D*));
    locCost[s1][s2] = localize(Cost[ ][ ], (DP,D*));
    locRoot[s1][s2] = localize(Cost[ ][ ], (DP,D*));
    float cost[s1][s2];
    int root[s1][s2];
    int prob[1..n] = Prob[1..n];
    barrier;
    for (k=0; k<=t-1;k++) {
        if (p <= t-1-k) {
            if (k>0) {
                float tcost[k*s1][s1];
                for (int q=1; q<=k; q++)
                    tcost[(q-1)*s1..q*s1-1][0..s1-1] =
                        cost<p+q>[0..s1-1][(k-q)*s1..(k+1-q)*s1-1];
                link cost[s1..(k+1)*s1-1][k*s1..(k+1)*s1-1]
                    to tcost[0..k*s1-1][0..s1-1];
            }
            for (int i=s1-1; i>=0; i--)
                for (int j=k*s1;j<=(k+1)*s1-1; j++)
                    MST(cost, prob, root, i,j);
        }
        barrier;
    }
    locCost[0..s1-1][0..s2-1]=cost[0..s1-1][0..s2-1];
    locRoot[0..s1-1][0..s2-1]=root[0..s1-1][0..s2-1];
}

MST(cost, prob, root, i, j)
{
    if (j <= i) cost[i][j] = 0;
    else if (j==i+1) {
        int gj = localToGlobal(Cost[ ][ ], j, 2);
        cost[i][j] = prob[gj];
        root[i][j] = gj;
    } else {
        int gi = localToGlobal(Cost[ ][ ], i+1, 2);
        int gj = localToGlobal(Cost[ ][ ], j, 2);
        int psum = +/prob[gi..gj];
        int bestCost = MAX;
        int bestRoot = -1;
        for (int r=i; r<j; r++) {
            rCost = psum + cost[i][r] + cost[r+1][j];
            if (rCost < bestCost) {
                bestCost = rCost;
                gj = localToGlobal(Cost[ ][ ], r, 2);
                bestRoot = gj;
            }
        }
        cost[i][j] = bestCost;
        root[i][j] = bestRoot;
    }
}

```

Figure 3: Parallel Pseudo-Code of Optimal Binary Search Tree

the parallel execution of the diagonals are controlled by the first `if` statement in the `for` loop of `k` (also see Figure 3). The second `if` statement is used to read the local data from other threads and put them in a temporary buffer called `tcost`[][]. Note that we use a link statement to link an extension of array `cost` to that buffer to facilitate writing the algorithm for function `MST()`. Since `n+1` is a multiple of `t`, `s1` is the same for all threads and `s2` equals to `(p+1)*s1`, where `p` is the thread identity. The `MST()` shown in Figure 3 is a bit more complicated than the sequential one, because we need to convert local indexes to the global ones by using function `localToGlobal()` in Peril-L.

Again, as the parallel pseudo-code for the Jacobi method showed, the parallel pseudo-code for the optimal binary search tree in Figure 3 is concise, rigorous and scalable. It contains no implementation details of message passing.

5 Conclusion

In this paper, we extended the Peril-L parallel pseudo-code language for writing parallel algorithms for CTA machines. The extensions we bring to Peril-L are: parallel thread grids, sub-array and partition description vectors, accessing size information and local variables of non-local threads, and linking of local variables. We demonstrated the effectiveness of the extended Peril-L by showing the parallel algorithms for the Jacobi method for solving differential equations and the dynamic programming method for finding the optimal binary search tree.

References

- [1] Wikipedia. Pseudocode. *Wikipedia: The Free Encyclopedia*, page <http://en.wikipedia.org/wiki/Pseudocode>, 2008.
- [2] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.
- [3] A. Gara, M.A. Blumrich, P. Coteus D. Chen, G. L.-T. Chiu, and et. al. Overview of the Blue Gene/L system architecture. *IBM Journal Research and Development*, 49(2/3):195–212, 2005.
- [4] M. Feldman. SiCortex get personnel. *HPCWire*, page <http://www.hpcwire.com/blogs/17911149.html>, 2008.
- [5] Galvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Addison Wesley, Pearson Education, Inc., 2008.
- [6] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Ed)*. Addison-Wesley, 2000.