XMT Applications Programming: Image Registration and Computer Graphics

Bryant Lee

April 28, 2006

Abstract

Historically, parallel programming has always been a difficult problem. Until now, the issue has been side-stepped by avoiding parallelism in favor of faster serial computers. However, major chip manufacturers like Intel and AMD agree that the clock race is now over and the only way to continue to make returns on increasing transistor density is to increase the number of processors on a chip. Parallel programming becomes a necessity, yet it remains an industry-wide stumbling block.

The Explicit Multithreading (XMT) framework has been advanced as a design for an easy to program parallel computer using the theoretical basis of PRAM algorithms. It is best summarized as a PRAM-On-Chip and could significantly ease the development of parallel applications. This paper describes how two particular real-world applications world were developed for XMT, medical image registration and fragment shaders for computer graphics.

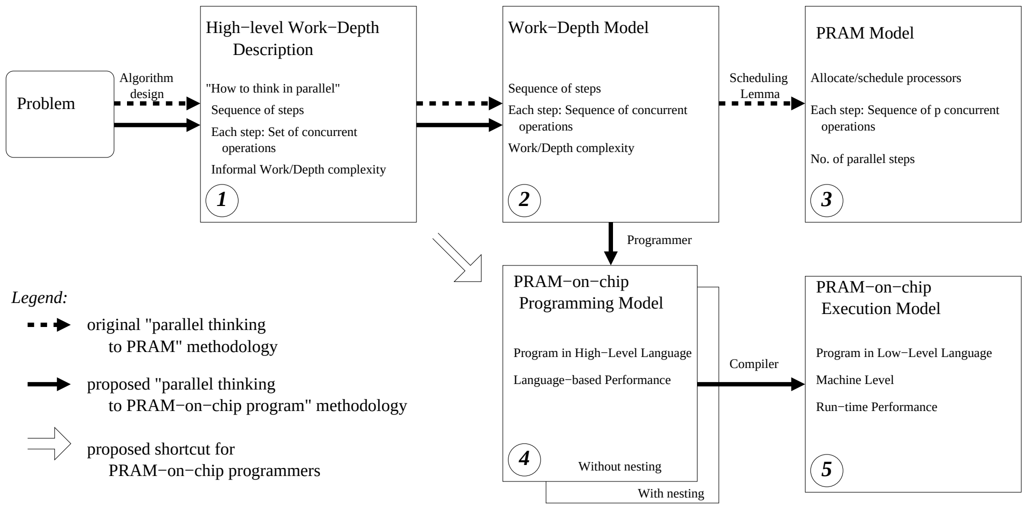
1 Introduction

Current methods of parallel programming include shared memory and message passing models and are typically coarse-grained. However, these methods often require the programmer to become involved in low-level details such as memory layout and architectural implementation, which makes the task especially difficult. Development of an easy method of parallel programming would be a breakthrough.

The Parallel Random Access Model (PRAM) might be the key to this breakthrough. The PRAM is an easy to use theoretical model that was developed in the 1980s and boasts the second largest knowledge base of algorithms next to serial RAM algorithms. It abstracts away hardware details and assumes processors are connected to a shared memory, of which they can access any element in constant time. Furthermore, the processors are synchronized and proceed in lock-step. Although the PRAM was widely accepted for some time as the best parallel algorithmic model and included in major textbooks [3, 7, 15], it eventually fell by the wayside due to doubts people held about the implementability of the PRAM model.

Despite this fact, some efforts were made towards implementation of PRAM algorithms. The NYU Ultracomputer was a pioneering effort in shared memory computing that made use of some elements of PRAM theory [1]. The Tera/Cray Multithreaded Architecture (MTA) seeks to hide latencies to memory by quick context switches between many hardware threads [18]. It has been suggested that the MTA's similarities to PRAM could allow efficient implementation of irregular graph problems [4]. The SB-PRAM is a computer that runs programs that are written very similarly to PRAM [10], and a prototype has been built. Finally, NESL is a functional language that allows easier expression of data parallel algorithms [5]. The research describe above has explored the practicality of PRAM, but PRAM itself has not found much use outside academia.

The XMT framework is a design for the architecture and programming model of a PRAM-On-Chip. With increasing transistor density it becomes possible to put large numbers of processing cores on a chip, which may allow some approximation of a PRAM. The distinguishing features of XMT are its larger bandwidth from the on-chip environment, lower latencies to the shared memory (e.g. on-chip shared cache), support for serial code, and support for parallel programs with low amounts of parallelism. Threads in XMT are defined by the language and not by an operating system. Furthermore, the threads are short-lived and follow



Problem
Algorithm design
High−level Work−Depth Description
"How to think in parallel"
Sequence of steps
Each step: Set of concurrent operations
Informal Work/Depth complexity
1
Work−Depth Model
Sequence of steps
Each step: Sequence of concurrent operations
Work/Depth complexity
2
Scheduling Lemma
PRAM Model
Allocate/schedule processors
Each step: Sequence of p concurrent operations
No. of parallel steps
3
Programmer
PRAM−on−chip Programming Model
Program in High−Level Language
Language−based Performance
4
Without nesting
With nesting
Compiler
PRAM−on−chip Execution Model
Program in Low−Level Language
Machine Level
Run−time Performance
5
Legend:
original "parallel thinking to PRAM" methodology
proposed "parallel thinking to PRAM−on−chip program" methodology
proposed shortcut for PRAM−on−chip programmers

Figure 1: Methodology for Developing PRAM-On-Chip Programs in view of the Work-Depth Paradigm for Developing PRAM algorithms. From [20].

Independence of Order Semantics (IOS). Each thread executes at its own pace and any order of interactions among threads is valid.

XMT has been used for various algorithmic problems including breadth-first search, sparse matrix-vector multiplication, and sorting [20]. A VHDL gate-level simulation was also developed [8]. The current paper describes two further applications: a medical image registration program and a fragment shader for computer graphics.

Image registration is used to find the spatial transformation that will align two similar images. For example, it can be used to overlay a CT and PET image of the same organ that were taken at different times, which is useful for creating synthetic images.

The second application considered, fragment shading, is used in current graphics processing units (GPUs). The fragment shader computes pixel colors from information passed to it by previous stages in the graphics pipeline. Current fragment shaders are sometimes difficult to program and have poor performance on some types of tasks. XMT could eliminate both these concerns.

First the XMT programming model and architecture is described in section 2. Section 3 describes the image registration application. Section 4 describes the application to fragment shading.

2 XMT Programming Model and Architecture

We first describe a methodology for transitioning from parallel algorithmic thinking to producing an implementable program. We explain both the XMT programming model and XMT architecture. We then provide simple examples of XMT programs.

2.1 From Parallel Algorithmic Thinking to Parallel Programming

A methodology for programming XMT has been advanced [20] and describes how to develop the high-level idea of a program into implementation. Figure 1 depicts this process. The sequence of parallel algorithmic thinking to produce a traditional PRAM algorithm is $1 \rightarrow 2 \rightarrow 3$. On the other hand, the process of producing a program for XMT follows the stages $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$. In some cases, stage 2 can be skipped due to XMT's support for certain constructs like nested parallelism that allow a direct transition of $1 \rightarrow 4$.

PRAM Model The parallel random access machine (PRAM) is an extension of the RAM model [2] to the case including multiple processors. The PRAM consists of p synchronous processors and a global shared memory accessible in unit time from each of the processors. The only means of interprocessors communication is through the shared memory.

Work-depth Model The work-depth methodology was introduced in [19] as a means of designing parallel algorithms. It is a useful and general framework for designing parallel algorithms and balances the aspects of depth, the amount of time an algorithm would take if infinite parallel hardware was available, and work, the total operations used. The High-Level Work-Depth (HLWD) methodology is used to describe a parallel algorithm informally. A HLWD description consists of a succession of parallel rounds, each being a set of instructions to be performed concurrently. The Work-Depth model is slightly lower-level and requires the concurrent operations in each time step to be ordered. The Work-Depth model is formally equivalent to the PRAM. The equivalence proof follows Brent's scheduling principle that was introduced in [6].

PRAM-on-chip (XMT) Programming Model In XMT, execution alternates between serial and parallel execution modes. Parallel execution is started through the spawn instruction, which creates a user-specified number of virtual threads that all have the same code. Each thread has a unique thread ID, which is accessed by the variable \$. Also, threads can use the prefix-sum (ps) instruction to perform an atomic fetch and increment to a base variable. The base variable of a ps is a special register in the XMT hardware. If B is the base variable, and I is the increment, then the result of the code $ps(I, B)$ is that I has the original value of B and B is set to $B + I$. For example, the following code compacts the non-zero elements of T into S:

```
spawn(0, n) {
    int increment = 1;

    if(T[$] != 0) {
        ps(increment, B);
        S[increment] = T[$];
    }
}
```

The ps instruction is supported by special hardware units that can combine ps calls into a multi-operand ps operation. This allows fast inter-thread synchronization.

In addition, there is a prefix-sum to memory (psm) instruction that operates the same as the ps instruction except the base variable is a memory location. This instruction is executed by queued updates to the memory location rather than by special hardware.

Nested spawning is allowed through use of the sspawn instruction, which allows a thread to spawn a new thread.

PRAM-on-chip (XMT) Execution Model A bird eye's view of XMT is presented in Figure 2. A number of (say 1024) Thread Control Units (TCUs) are grouped into (say 64) clusters. Clusters are connected to the memory subsystem by a high-throughput, low-latency interconnection network; they also interface with specialized units such as prefix-sum unit and global registers. A hash function is applied to memory addresses in order to provide better load balancing at the shared memory modules. An important component of a cluster is the read-only cache included at cluster level; this is used to store values read from memory by a TCU and also holds the values read by prefetch instructions. The memory system consists of memory modules each having several levels of cache memories. In general each logical memory address can reside in only one memory module, alleviating cache coherence problems. This explains why only read-only caches are used at the clusters. The Master TCU runs serial code, or the serial mode for XMT. When it hits a Spawn command it initiates a parallel mode by broadcasting the same SPMD parallel code segment to all the TCUs. As each TCU captures its copy, it executes it is based on a thread-id assigned to it. A separate distributed hardware system, reported in [17] but not shown in figure 2, ensures that all the thread id's mandated by the current Spawn command are allocated to the TCUs. A sufficient part of this allocation is done dynamically to ensure that no TCU needs to execute more than one thread id, once another TCU is already idle.

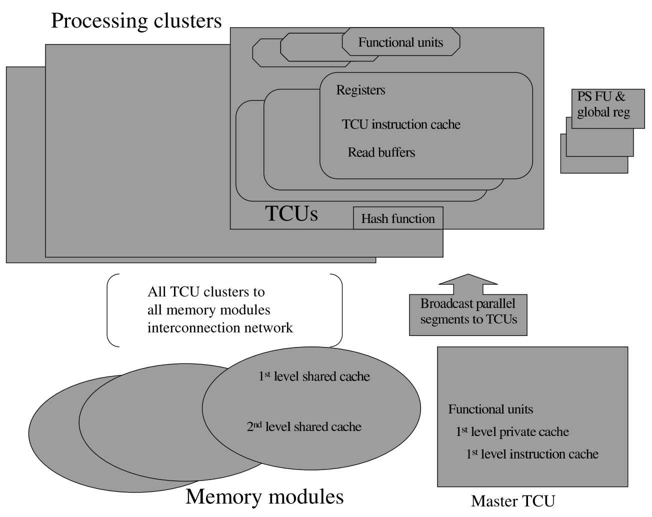


Figure 2: An overview of the XMT PRAM-on-chip Architecture.

```
SUM(A,n)
 If n = 1 then sum = A[1]; exit
 For 1 <= i <= n / 2 pardo
          B[i] = A[2i - 1] + A[2i]
 Call SUM(B,n/2)
```

(a)

```
For 1 <= i <= n pardo     // B is a 1D array
        B[n-1 + i] = A[i]// model of a tree
For h = logn to 1 do
        For 2^(h-1) <= i < 2^h pardo
                B[i] = B[2i - 1] + B[2i]
sum = B[1]
```

(b)

Figure 3: The Summation Algorithm. (a) A High-Level Work Depth presentation. Pairs of values of A are summed up and stored into array B, followed by a recursive call on array B. (b) A Work-Depth description

2.2 Examples: Summation, Prefix-sums, and Breadth-first Search

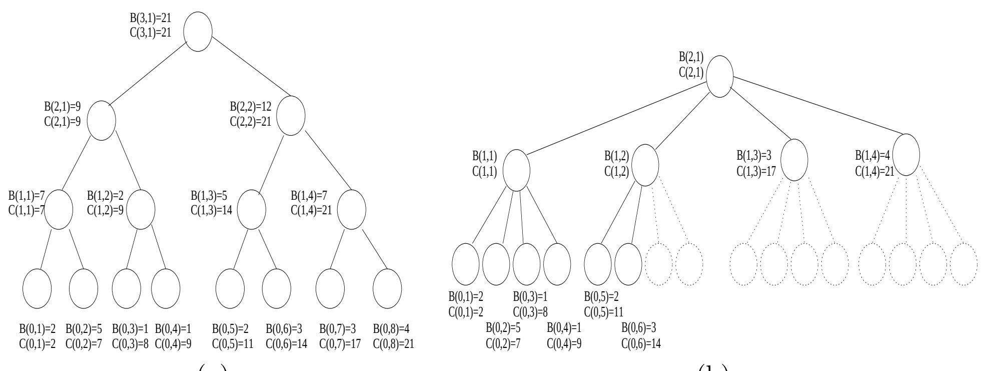
We provide some brief examples of programming in XMT. Actual code for the examples is provided in table 6.

2.3 Summation

Consider the problem of computing the sum of n numbers. Given as input an array A of size n the output provides the sum of its values. Developing a parallel program for this simple problem is presented next as an example for the methodology of the previous section. Progressing through the models is presented. A High-Level Work-Depth description of the algorithm is presented in figure 3.a. A non-recursive Work-Depth presentation of this algorithm can be derived from it, as presented in figure 3.b.

The tree of values is represented using a unidimensional array. In the general case of a complete k-ary tree, we store the root at element 0, followed by the k elements of the first level, listed from left to right, then the k^2 elements of second level etc. The array is densely packed, with no gaps, thus (a) the children of node i are at indices $k*i+1, k*i+2, \ldots, k*i+k$ and (b) the parent of node i is at index $\lfloor \frac{i-1}{k} \rfloor$. Note that this simple relationship between a node and its children is helpful for improving performance.

The conversion to the XMT code of figure 6.a is seen to be straightforward.



(a) (b)

Figure 4: (a) PRAM prefix-sums algorithm on a binary tree and (b) PRAM prefix-sums algorithm on a k-ary tree (k=4).

2.4 Prefix-sums

Prefix-sums is a basic routine underlying many parallel algorithms. Given an array $A[0..n-1]$ as input, let $prefix\_sum[j] = \sum_{i=0}^{j-1} A[i]$ for j between 1 and n and $prefix\_sum[0] = 0$.

Due to [14], the basic routine works in two stages each taking $O(\log n)$ time. The first stage is the Summation algorithm presented previously, namely the computation advances up a balanced tree computing sums. The second stage advances from root to leaves. Each internal node has a value $C(i)$, where $C(i)$ is the prefix-sum of its rightmost descendant leaf. The $C(i)$ value of the root is the sum computed in the first stage, and the $C(i)$ for other nodes is computed recursively. Assuming that the tree is binary, any right child inherits the $C(i)$ value from its parent, and any left child takes $C(i)$ equal to the $C(i)$ of its left uncle plus this child's value of sum. The values of $C(i)$ for the leaves are the desired prefix-sums. See figure 4.

The implementation of this algorithm in the PRAM-On-Chip Programming model is presented in figure 6.b using XMTC pseudocode. Similar to the Summation algorithm, we use a k-ary tree instead of a binary one. The two overlapped k-ary trees are stored using two one-dimensional arrays *sum* and $prefix\_sum$ by using the array representation of a complete tree as discussed in section 2.3.

The PRAM-On-Chip algorithm works by first advancing up the tree using a summation algorithm. Then the algorithm advances down the tree to fill in the array $prefix\_sum$. The value of $prefix\_sum$ is defined as follows: (a) for a leaf, $prefix\_sum$ is the prefix-sum and (b) for an internal node, $prefix\_sum$ is the prefix-sum for its leftmost descendant leaf.

2.5 Breadth-first Search

Given is an undirected graph $G(V, E)$, where the length of every edge in E is 1, and a source node $s \in V$; the *breadth-first search (BFS)* algorithm finds the lengths of the shortest paths from s to every node in V. An informal work-depth description of the parallel BFS algorithm can look as follows. Suppose that V, the set of vertices of the graph G, is partitioned into layers, where layer L_i includes all vertices of V whose shortest path from s includes exactly i edges. The algorithm works in iterations. In iteration i, layer L_i is found. Iteration 0: node s forms layer L_0. Iteration i, $i > 0$: Assume inductively that layer L_{i-1} has already been found. In parallel, consider all the edges (u, v) that have an endpoint u in layer L_{i-1}; if v is not in a layer L_j, $j < i$, it must be in layer L_i. As more than one edge may lead from a vertex in layer L_{i-1} to v, vertex v is marked as belonging to layer L_i by one of these edges using the arbitrary concurrent write convention. This ends an informal, high-level work-depth verbal description.

A pseudocode description of an iteration of this algorithm is given in figure 5.

A detailed implementation of this algorithm using the XMTC programming language is included in figure 6.c. To traverse an edge, threads use an atomic prefix-sum instruction on a special "gatekeeper" memory location associated with the destination node. All gatekeepers are initially set to 0. Receiving a 0 from the prefix-sum instruction means the thread was the first to reach destination node, and the newly discovered

```
for all vertices v in L(i) pardo
  for all edges e=(v,w) pardo
    if w unvisited
      mark w as part of L(i+1)
```

Figure 5: Pseudo-code of one iteration of the BFS algorithm.

neighbors are added to layer $L(i+1)$ using another prefix-sum operation on the size of $L(i+1)$. In addition, the edge anti-parallel to the one traversed is marked to avoid needlessly traversing it again (in the opposite direction) in later BFS layers.

3 Image Registration

An image registration program for XMT was developed based on a serial image registration program created by Dr. Carlos Castro-Pareja of the University of Maryland School of Medicine. In future work we will run the program on an XMT simulator to determine how fast image registration may be done on an XMT architecture [12]. The speedups for XMT will be recorded relative to a serial software implementation and also to a special purpose hardware implementation designed by Dr. Castro-Pareja and collaborators. We believe that the XMT implementation may be: (i) significantly faster than the serial software implementation, and (ii) be sufficiently competitive with the special hardware implementation to merit using XMT due to its being general-purpose.

3.1 Serial Registration Algorithm

Problem definition: the input is one 3D image that is the floating image (FI) and another 3D image that is the reference image (RI). The typical size of images used is 128^3 up to 512^3. The images are two different images of the same object, taken at different angles, offsets, moments in time, and so forth. The image registration problem is to find one global transformation and many local transformations such that if these transformations were applied to FI, then FI could be laid on top of RI and the images would match up well.

Figure 7 describes the algorithm at a high level. First, the program performs global registration, finding the global transformation of FI that makes FI fit best with RI. The global transformation sought is always *rigid*, meaning it is simply a translation and rotation.

After applying the best global transformation to FI, the program performs local registration on FI. Local registration finds different transformations for different regions of FI that make FI fit best with RI. In the literature, this is also known as *elastic* registration because the voxels, the equivalent of pixels in 3D, of FI do not all have the same transformation so the effect is as if FI is being pulled in different directions.

Global Registration Figure 8 shows the global registration algorithm. On each iteration, the algorithm applies a candidate transformation Tx to FI to create FI' and measures the fit of FI' to RI through some process. If the fit is optimal, the algorithm quits, otherwise another candidate transformation is generated and the process is repeated. The first candidate transformation is user specified.

Local Registration The local transformations are modeled via control boxes (CBs). The CBs are sets voxels forming non-overlapping cubes of uniform size that cover FI. Each CB has its own transformation, and this transformation applies to all the voxels within its boundaries. Thus, one CB represents one local transformation.

In local registration, the best transformation for each CB is found. On each CB, we use the same high-level algorithm used for global registration. Figure 9 shows this. The following is done for each CB_i: take the transformation stored in CB as the first candidate transformation Tx. At each iteration, Tx is applied to CB_i to create ${CB_i}'$ and then measure the fit of ${CB_i}'$ to RI, considering only the area of RI overlapping

(a) k-ary Tree Summation

```
/* Input: N numbers in sum[0..N-1]                                              *
 * Output: The sum of the numbers in sum[0]                                      *
 * The sum array is a 1D complete tree representation (See Summation section)   */
level = 0;
while(level < log_k(N) ) { // process levels of tree from leaves to root
        level++;
        spawn(current_level_start_index, current_level_end_index) {
                int count, local_sum=0;
                for(count = 0; count < k; count++)
                        temp_sum += sum[k * $ + count + 1];
                sum[$] = local_sum;
        }
}
```

(b) k-ary Tree Prefix-Sums

```
/* Input: N numbers in sum[0..N-1]                                              *
 * Output: the prefix-sums of the numbers in                                     *
 *  prefix_sum[offset_to_1st_leaf..offset_to_1st_leaf+N-1]                       *
 * The prefix_sum array is a 1D complete tree representation (See Summation)   */
kary_tree_summation(sum); // run k-ary tree summation algorithm
prefix_sum[0] = 0; level = log_k(N);
while(level > 0) { // all levels from root to leaves
        spawn(current_level_start_index, current_level_end_index) {
                int count, local_ps = prefix_sum[$];
                for(count = 0; count < k; count++) {
                        prefix_sum[k*$ + count + 1] = local_ps;
                        local_ps += sum[k*$ + count + 1]; }
        }
        level--;
}
```

(c) Breadth-First Search

```
/* Input: Graph G=(E,V) using adjacency lists (See Programming BFS section)    *
 * Output: distance[N] - distance from start vertex for each vertex            *
 * Uses: level[L][N] - sets of vertices at each BFS level.                     */
//run prefix sums on degrees to determine position of start edge for each vertex
start_edge = kary_prefix_sums(degrees);
level[0]=start_node; i=0;
while (level[i] not empty) {
    spawn(0,level_size[i] - 1) {  // start one thread for each vertex in level[i]
        v = level[i][$]; // read one vertex
        spawn(0,degree[v]-1) {   // start one thread for each edge of each vertex
            int w = edges[start_edge[v]+$][2]; // read one edge (v,w)
            psm(gatekeeper[w],1);//check the gatekeeper of the end-vertex w
            if gakeeper[w] was 0 {
                 psm(level_size[i+1],1);//allocate one entry in level[i+1]
                 store w in level[i+1]; }
        }
    }
    i++;
}
```

Figure 6: Implementation of some PRAM algorithms in the XMT PRAM-on-chip framework to demonstrate compactness.

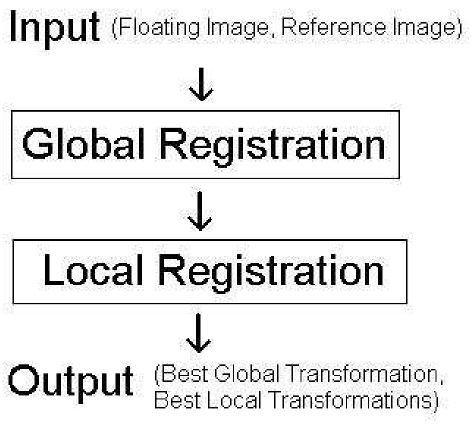

Input (Floating Image, Reference Image)
Global Registration
Local Registration
Output (Best Global Transformation, Best Local Transformations)

Figure 7: High level description of the serial algorithm.

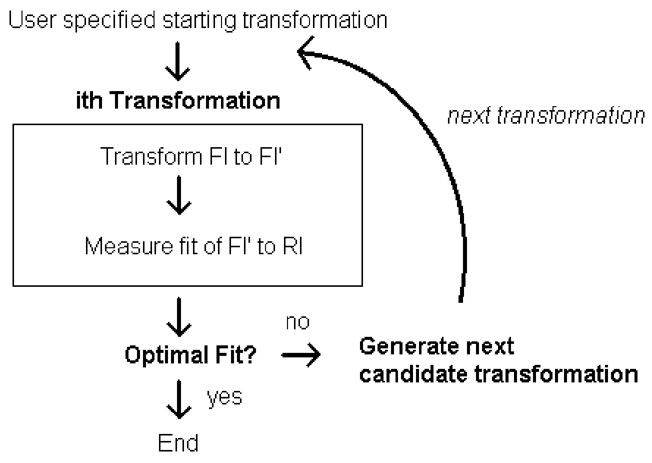


Figure 8: Global registration algorithm.

CB_i'. If the fit is good enough, the algorithm quits, otherwise another candidate transformation is generated and the process is repeated. The best transformation is stored in CB_i.

3.2 Serial Registration Program

Having introduced the main ideas of the algorithm, we proceed to describe the program at a lower level of detail. The program has two parts, global registration and local registration, as was described previously.

Global Registration On each iteration, the program applies a candidate transformation to FI, measures fitness of FI' to RI, and ends if the transformation is good enough. Fitness is measured in the following way: a mutual histogram of RI and FI' is created; this is a 2-dimensional histogram with RI intensity on one axis and FI' intensity on the other. From the mutual histogram, a straightforward calculation gives the mutual information (MI), which is a single number that grades how good the fit is between RI and FI'. The MI is fed into an implementation of Simplex, a well-known algorithm for finding the minimum of a function of more than one variable. Simplex decides either that this is the best MI possible (within a tolerance) or generates another candidate transformation. The process is repeated.

The step of transforming FI to FI' must be explained in more detail because this will be focused on in the parallelization of the program to be described later. Transforming is done backwards. We do not iterate over the voxels of FI and transform each one in turn as might be expected; instead, we iterate over the voxels of RI and for each voxel r_i, find the point p_i in FI that would fall on top of that voxel if FI were placed under the transformation. The point p_i defines a precise floating point spot, which does not necessarily fall on a voxel of FI (voxels should be thought of as falling on integer coordinates). So, for each voxel r_i of RI, we have a point p_i in FI, and we want to fill in the mutual histogram of RI and FI'. The 8 voxels in FI that are nearest to p_i, call them $f_{i_0} \dots f_{i_7}$, are given weights $w_{i_0} \dots w_{i_7}$ according to how close they are to p_i. See Figure 10. The mutual histogram is updated as follows: $MutualHist[intensity(r_i)][intensity(f_{i_K})] += w_{i_K}$ where K in $0 \dots 7$.

Local Registration It has been shown empirically that best results are obtained not by starting with a constant number of CBs and running registration on each of them, but instead by starting with a small number of CBs and gradually ramping up to the desired number of CBs. On each iteration, the best transformation for each CB is found through registration matching on every CB. Then granularity is increased by doubling the number of CBs in each dimension. Iterate until the desired number of CBs is reached. At this point, local registration is finished, because we have as many CBs as we desire and each has the best transformation possible.

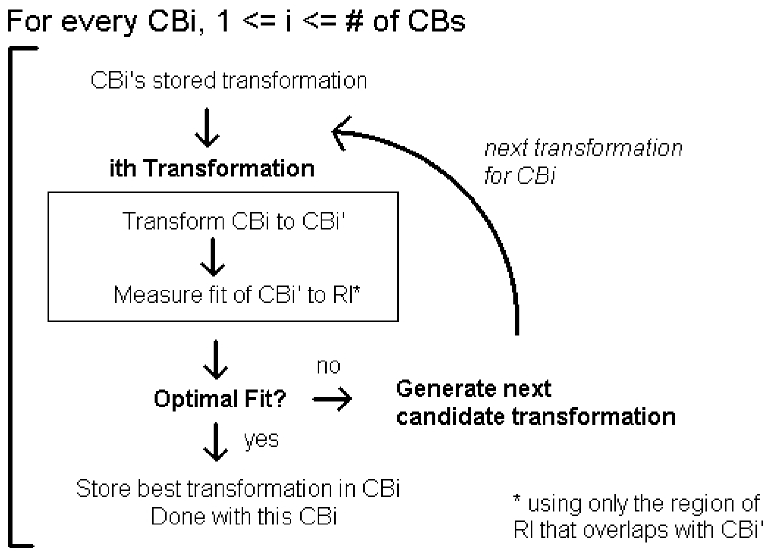

For every CBi, 1 <= i <= # of CBs
CBi's stored transformation
ith Transformation
Transform CBi to CBi'
Measure fit of CBi' to RI*
next transformation for CBi
Optimal Fit?
no
Generate next candidate transformation
yes
Store best transformation in CBi Done with this CBi
* using only the region of RI that overlaps with CBi'

Figure 9: Local registration algorithm.

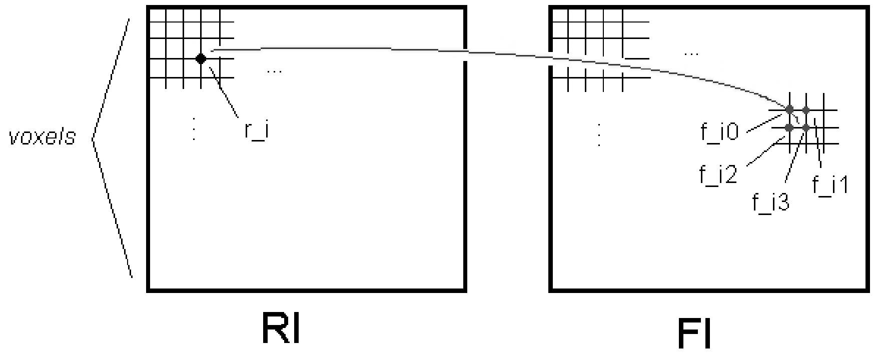


Figure 10: Finding the point in FI that corresponds to a voxel in RI.

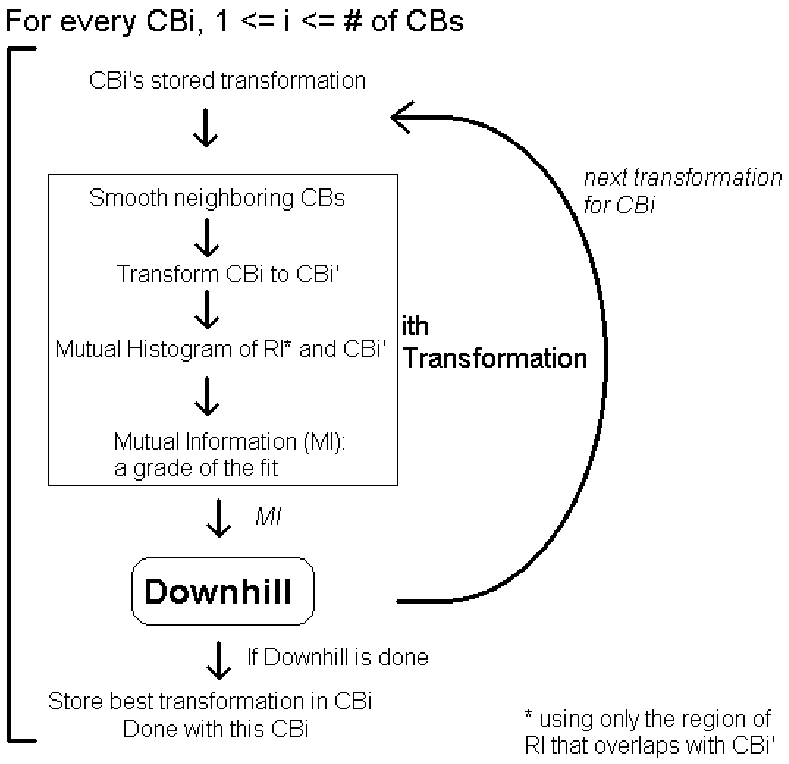

For every CBi, 1 <= i <= # of CBs
CBi's stored transformation
Smooth neighboring CBs
Transform CBi to CBi'
Mutual Histogram of RI* and CBi'
Mutual Information (MI): a grade of the fit
ith Transformation
next transformation for CBi
MI
Downhill
If Downhill is done
Store best transformation in CBi
Done with this CBi
* using only the region of RI that overlaps with CBi'

Figure 11: Registration matching on every CB.

By performing registration matching on every CB, best transformation for each CB is found. Figure 11 shows the program. The following is done for each CB_i: the transformation stored in CB_i is taken to be the first candidate transformation Tx. Next, a new step called *smoothing* is introduced. If neighboring CBs have transformations very different from that of the candidate transformation, we smooth them (described in further detail later). Aside from smoothing, the program is a straightforward implementation of the high-level local registration algorithm. Tx is applied to CB_i , a mutual histogram of $CB_i{}'$ and RI is created (using only the area of RI overlapping CB_i'), the MI is calculated, and the MI is fed into an implementation of Downhill, an algorithm for finding the minimum of a function which is different from but analogous to Simplex. If Downhill decides this is the best MI possible then we store the best transformation in CB_i and iterate to the next CB_i.

To keep the local transformations from being too disjointed from one another we enforce a max and min difference between transforms of adjacent CBs. Whenever a new candidate transformation is chosen for CB_i, the transformations of the 8 adjacent CBs are checked to see if any is too different from the candidate transformation. Suppose there is a CB_j with a transform that is too different. Then CB_j's transformation is changed to be within the desired bounds. Since CB_j's transformation has changed, the CBs that are adjacent to CB_j must now also be checked to not be too different from CB_j's new transformation. The propagation continues in this manner. It is important to note that the only thing being changed in this step is a CB's stored transformation, no operations take place on voxels and nothing is recalculated. Furthermore, while it is not necessary to mention here the exact method of propagation, it should be said that each CB is checked and changed at most once, so that, in particular, it is impossible for circularity to occur.

This concludes the discussion of the serial registration program. The pseudo-code of the program is provided in appendix A.

```
1. PARDO i, i = each voxel r_i in RI
2.        Find the point p_i in FI that falls on the voxel r_i in RI when
          FI is under the transformation
3.        Get nearest 8 voxels in FI and weight them according to how close
          they are to p_i
4.        Accumulate info of the 8 voxels into mutual histogram (MH), using
          ps instruction to control concurrent access to MH
```

Figure 12: Pseudo-code for transforming FI to FI' and creating the mutual histogram of RI and FI'.

3.3 Parallelization

The XMT image registration program was developed to have exactly the same functionality as the serial program provided by Dr. Castro-Pareja. The code was ported from serial C++ to XMTC, but the algorithm is unchanged. This is important to emphasize given that many design choices are made when developing any image registration program. For example, image similarity can be measured by different mathematical calculations than mutual information (some other well-researched similarity measures are Cross-Correlation, Pattern Intensity, and Mean Square Difference of Intensities). Likewise, different optimization algorithms than Simplex and Downhill could be used, such as Powells Method or Simulated Annealing. However, we have adhered to the original program provided by Dr. Castro-Pareja so that, when the XMT program is run through an architectural simulator, the speedup obtained will reflect only a difference in architecture.

Several steps of the program were parallelized. Most important among these is the parallelization of the process of transforming FI based on a transformation Tx. This process is used in global and local registration. In global registration, it comes into play in transforming FI to FI' and creating the mutual histogram of RI and FI'. In local registration, it involves the two steps transforming the CB_i to CB_i' and creating the mutual histogram of RI and CB_i'. The transformation process consumes the vast majority of computation time in the serial program and so parallelization of this step is expected to produce the largest speedup.

We describe how to parallelize the transformation process for use in global registration. The parallelization for local registration, in transforming the CB_i to CB_i' and creating the mutual histogram of RI and CB_i', is analogous.

Instead of iterating over the voxels of RI, we handle all the voxels of RI in parallel. In other words, we spawn one thread for each voxel r_i. Figure 13 illustrates how multiple transformations are performed at once in comparison with the serial program in which only one transformation is done at a time. Each thread does the following: finds the point p_i in FI that would fall on r_i if FI were placed under the transformation; finds the 8 nearest voxels to p_i in FI, call them $f_{i_0} \dots f_{i_7}$, and gives them weights $w_{i_0} \dots w_{i_7}$; and writes into the mutual histogram in the normal manner. Concurrent access to the histogram is handled by using the ps instruction. The pseudo-code is shown in figure 12.

Another major, but less significant, process that was parallelized was the process of increasing the granularity of local registration by doubling the number of CBs in each dimension. Since the number of CBs has increased, the new CBs must have their value initialized by interpolation from the old CBs. The interpolations are strictly local operations, and so they can all be done at once.

The last remaining process of note to be parallelized is subsampling. In the image registration program, as a preprocessing step after rigid registration and before local registration, the original RI and FI images may be reduced in size via subsampling in order to make the registration faster with some trade-off in accuracy. Each dimension of the image is reduced by a power of 2. The new voxel values are determined using interpolation, so this process is similar to the process for doubling the CBs except in reverse because the image is shrunk rather than enlarged. The interpolations are all done at once in a parallel operation.

In future work, the XMT image registration program will be run on an architectural simulator of XMT and timing results obtained [12].

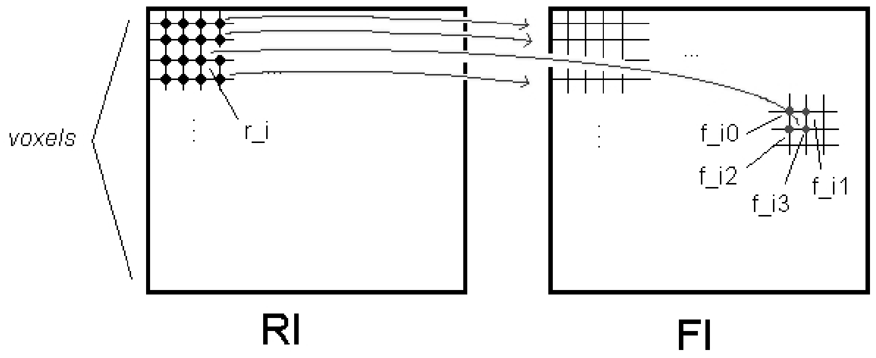


Figure 13: In parallel: finding the point in FI that corresponds to a voxel in RI.

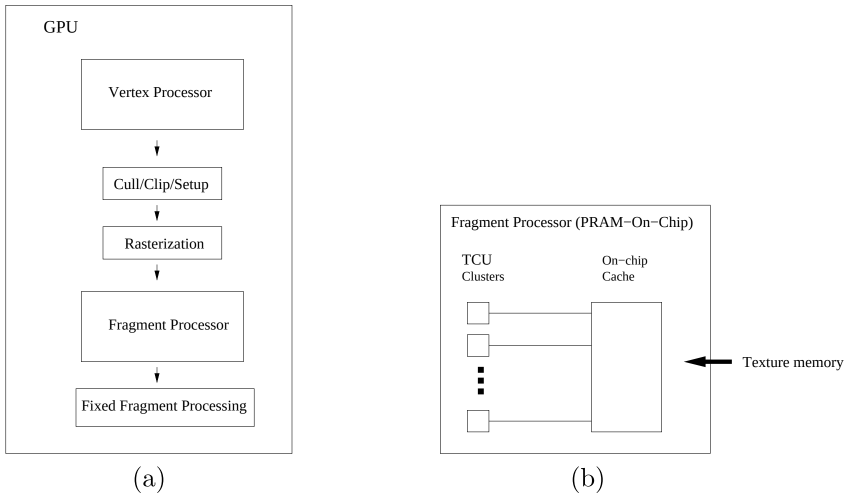


Figure 14: (a) The standard graphics pipeline, and (b) the use of XMT as a fragment shader.

4 Fragment Shaders for Graphics Processing

In this section, we describe how the XMT can be used in the important application domain of graphics processing. Graphics processing units (GPUs) are commonly found in consumer systems and can be used for visualization, animated movies, or 3D games. Modern GPUs are stream-based architectures in which graphical model data flows through a graphics pipeline. The use of streaming allows high performance but restricts the kinds of computations that can be done efficiently.

Graphics hardware has become more flexible as some of the previously fixed processing of the pipeline has become programmable. However, the new programmable processors that have been added are usually also stream-based, and for this reason are quite constrained. With respect to graphics processing, streaming has two major disadvantages i) limited and difficult programming and ii) poor performance on computing tasks that do not translate well to stream algorithms.

4.1 The Graphics Pipeline and XMT

An overview of the standard GPU pipeline is shown in figure 14.a. The graphical model data input by the user first passes through the vertex processor, which transforms the vertices of the model and also performs lighting calculations per vertex. The vertex processor is often a programmable stream processor. The vertices are then processed in the Cull/Clip/Setup stage, where some of the geometry that falls outside of the visible scene is removed. The output of this stage are the triangles that make up the scene. The triangles are then rasterized into flat polygons suitable for display on a 2D monitor, and the output of the rasterization stage is the individual fragments that will determine the final displayed image. Each fragment corresponds to a single pixel and carries color, texture, and other information for that pixel. These fragments then flow through a programmable, stream-based fragment processor to produce final pixel colors. Some fixed fragment processing may also be done, which is represented by the last stage of the pipeline.

XMT could be used for fragment processing instead of using a stream-based fragment processor. An overview of the XMT fragment processor is shown in figure 14.b. It would allow the programmer to write fragment shaders (programs run on the fragment processor) in the XMT programming model discussed in previous sections rather than in the streaming model.

The use of XMT would allow more general algorithms that do not translate well to streaming. For example, branching and looping is only emulated on a stream-based processor by using multiple passes. Also, although stream-based fragment processors have large numbers of processor, the processors can only communicate with each other by writing data to global memory on a first pass and using a second pass to read the data.

XMT could also be used as a vertex processor, but we decided to focus on the fragment processor because it is more often the bottleneck in the modern graphics pipeline.

We describe our current work on three different types of fragment shaders: procedural, texture mapping, and shaders for particle simulation. The shaders are being developed to allow a potential XMT fragment processor for OpenGL, a popular graphics API.

4.2 Basic Example: Simple Procedural Shader

A procedural shader computes pixel colors from fragment data without any accesses to texture memory. We present the example of a shader that produces a "brick wall" pattern by coloring some pixels a mortar color and other pixels a brick color.

The XMT fragment shader takes as input an array of fragments that has flowed in from previous stages of the pipeline. Each fragment has several attributes. For the brick shader, each fragment has an x and y position. The brick shader algorithm is an embarrassingly parallel algorithm where one thread is spawned for each fragment. Each thread computes whether the fragment lands on a brick or on mortar based on the fragment's x and y position, decides the pixel color based on this, and writes the pixel color as an attribute of the fragment. We have developed a shader of this type for XMT.

For simple procedural shaders such as this, the goal of XMT is to match the performance of the streaming implementation. The embarrassingly parallel brick shader algorithm is equally well adapted for streaming or XMT.

4.3 Texture Mapping Shader

Texture mapping is the default shader used in graphics processing. Fragments have an associated texture and x and y coordinates indicating the location in the texture that should be used to color the fragment. The algorithm is again embarrassingly parallel. The primary step involves spawning one thread for each fragment. The thread fetches the data from location (x, y) in the texture and uses the data to compute the pixel color. More complicated algorithms can involve interpolating between several textures, but the algorithmic pattern is the same. We have implemented a texture mapping shader with the same main functionality as found in GPUs. As with the brick shader, a texture mapping shader is well suited to either a streaming architecture or for XMT.

4.4 Particle Simulation

The power of stream-based GPUs is recognized for the types of shaders mentioned above, but researchers have also tried to extend the use of GPUs to other kinds of problems. Several more general problems have been shown to be solvable with a GPU, but only with complicated multipass algorithms that require exploiting low-level details of the graphics processing pipeline [9, 11, 16]. On the other hand, XMT is suitable for general purpose computing.

The problem of particle simulation with inter-particle collision detection is one that has been implemented on a streaming GPU but may be better suited to XMT. Particle simulation is used in computer graphics to generate effects such as fire, smoke, and fluids. It is also used for physically based dynamics simulation. Kipfer, Segal, and Westermann [11] developed a streaming GPU algorithm for the problem. The input is a set of particles and the velocities and forces that define their motion. The algorithm proceeds in a series of time steps. At the beginning of a time step, an embarrassingly parallel algorithm is used to move each particle without checking or responding to collisions. Then, particles are sorted on their location in 3D space by using GPU bitonic sort. Then, in parallel, collisions are identified in the sorted set of particles and are resolved.

An advantage that XMT has in this case is the ability to use hierarchical spatial data structures rather than requiring a sort on the particles. For example, an octree decomposition of space allows a fast search for neighbors. The limited functionality of streaming GPUs prevents use of data structures of this kind. A particle simulation engine for XMT is currently being developed. This, as well as other work discussed in the current section is joint with M. Olano.

Timing results for running the fragment shaders on XMT and comparison to GPUs will be presented in future work [13].

5 Conclusion

XMT leverages the already mature PRAM theory to make parallel programming easy for many general problems. We have developed two real-world applications for XMT, a medical image registration program and fragment shaders for graphics processing. In future work, we will obtain timing results for the applications by running them on an architectural simulator.

APPENDIX

A Pseudo-code of Serial Image Registration Program

```
// Global Registration
1. Tx = initial transformation          //Tx is the candidate transformation
2. Do until Simplex is done
3.      for i, i = each voxel r_i in RI
4.              Find the point p_i in FI that falls on the voxel r_i in RI
                when FI is under the transformation
5.              Get nearest 8 voxels in FI and weight them according to
                how close they are to p_i
6.              Accumulate info of the 8 voxels into MH
7.      Calculate mutual information from MH
8.      Plug mutual information into Simplex
9.      Tx = get next transformation from Simplex

// II. Local Registration
10. for res, 1 <= res <= resolutions_desired
11.     for i, i = each CBi
12.             Tx = get transformation stored in CBi
13.             Do until Downhill is done
14.                     Change the transformations of neighboring CBs if they
                        are too different from the transformation of CBi
15.                     for i, i = each voxel r_i in RI that falls in CB
16.                             Find the point p_i in FI that falls on the voxel
                                r_i in RI when FI is under the transformation
```

```
17.                             Get nearest 8 voxels in FI and weight
                                them according to how close they are to p_i
18.                             Accumulate info of the 8 voxels into MH
19.                     Calculate mutual information from MH
20.                     Plug mutual information into Downhill
21.                     Tx = get next transformation from Downhill
22.             CBis transformation = best Tx
23.     Double the number of CBs in each dimension
```

References

[1] G.S. Almasi and A. Gottlieb. Highly Parallel Computing. Benjamin Cummings, 1994.

[2] A. V. Aho and J. D. Ullman. *Foundations of Computer Science.* W. H. Freeman & Co., New York, NY, USA, 1994.

[3] S. Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, 1988.

[4] D.A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. The 33rd International Conference on Parallel Processing, pp. 547-556, 2005.

[5] G.E. Blelloch. Programming parallel algorithms. Communications of the ACM, 39(3), 1996.

[6] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201-206, 1974.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[8] P. Gu and U. Vishkin. Case study of gate-level logic simulation on an extremely fine-grained chip multiprocesor. Journal of Embedded Computing, Special Issue on Embedded Single-Chip Multicore Architectures and Related Research from System Design to Application Support. To appear in 2006.

[9] M.J. Harris, W.V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware*, Eurographics Assocation, 2003, pp. 92-101.

[10] J. Keller, C.W. Kessler, and J.L. Traff. Practical PRAM Programming. Wiley, New York, 2000.

[11] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A GPU-based particle engine. In *Graphics Hardware.* ACM SIGGRAPH/Eurographics, ACM Press, 2004.

[12] B. Lee, C. Castro-Pareja, and U. Vishkin. Real-time 3-D image registration on a fine-grained parallel computer. In preparation.

[13] B. Lee, Y. Wang, M. Olano, and U. Vishkin. Generalized parallel fragment shading. In preparation.

[14] R.E. Ladner and M.J. Fischer. Parallel prefix computation. Journal of the ACM, 27(4): 831-838, 1980.

[15] U. Manber. *Introduction to Algorithms - A Creative Approach.* Addison-Wesley, 1989.

[16] K. Moreland and E. Angel. The FFT on a GPU. In *Graphics Hardware*, Eurographics Assocation, 2003, pp. 112-119.

[17] D. Naishlos, J. Nuzman, C.-W. Tseng, and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. *TOCS*, 36 (2003), 521-552 (Special Issue of SPAA2001).

[18] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K.S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. Conference on High Performance Networking and Computing, 1998.

[19] Y. Shiloach and U. Vishkin. An $O(n^2 logn)$ parallel max-flow algorithm. Journal of Algorithms, 3(2): 128-146, 1982.

[20] U. Vishkin, G. Caragea, and B. Lee. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform (submitted). Eds. J. Reif and S. Rajasekaran, *Handbook of Parallel Computing*.