A

PLDS: Partitioning Linked Data Structures for Parallelism

Min Feng, Changhui Lin, and Rajiv Gupta, University of California, Riverside

Recently, parallelization of computations in the presence of dynamic data structures has shown promising potential. In this paper, we present PLDS, a system for easily expressing and efficiently exploiting parallelism in computations that are based upon dynamic linked data structures. PLDS improves the execution efficiency by providing support for data partitioning and then distributing computation across threads based upon the partitioning. Such computations often require the use of speculation to exploit dynamic parallelism. PLDS supports a conditional speculation mechanism that reduces the cost of speculation. PLDS can be employed in context of different forms of parallelism, which together cover a wide range of parallel applications. PLDS provides easy-to-use compiler directives, using which programmers can choose from amongst variety of data partitionings, distribute computation across threads in a partitioning sensitive fashion, and use conditional speculation when required. We evaluate our implementation of PLDS using ten benchmarks, of which six are parallelized using speculation. PLDS achieves 1.3x–6.9x speedups on an 8-core machine.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Language, Algorithms, Performance

Additional Key Words and Phrases: Linked data structures, parallelization, parallel programming, speculation, partitioning

ACM Reference Format:
Feng, M., Lin, C., and Gupta, R. 2012. PLDS: Partitioning Linked Data Structures for Parallelism. ACM Trans. Architec. Code Optim. V, N, Article A (January YYYY), 20 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Computation distribution is crucial to the performance of parallel applications. Efficient parallel execution requires exploiting locality of data references in the process of computation distribution. Many sequential programs consecutively execute the computations that share data to improve data locality. However, when parallelizing these programs, we should not distribute consecutive computations to different threads contemporaneously since it will cause data contention and thereby harm performance.

Data partitioning based computation distribution [Anderson and Lam 1993] has been proposed to improve the performance of parallel programs in distributed systems community. The idea is to first partition the data, then assign partitions to threads, and finally assign computations to threads such that the thread that owns the data required by a computation performs the computation. Implementation of this strategy is not a trivial task. Developers need to write code for: partitioning and assigning the data to threads; and distributing computation among threads based on the partitioning. They also need to enforce synchronization between computations from different partitions. A few programming models have been proposed to explore data and com-

This research is supported by NSF grants CCF-0963996 and CCF-0905509 to UC Riverside.
Author's addresses: M. Feng, C. Lin and R. Gupta, Computer Science and Engineering Department, University of California, Riverside, CA; email: mfeng@cs.ucr.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

putation distribution. Most of them focus on array-based parallel programs [Kennedy and Allen 2001; Rogers and Pingali 1989; Consortium 2005; Dotsenko et al. 2004; Hilfinger et al. 2005; Chamberlain et al. 2007; Charles et al. 2005]. The Galois system [Kulkarni et al. 2008] employs data partitioning for irregular JAVA applications. It is designed for exploiting worklist-based parallelism where an iterative program processes work items from a worklist. Galois requires programmers to specify the relationship between method calls and how to undo modifications for shared data structures. Compared to C++ programs, JAVA programs are less challenging since JAVA supports more OO features (e.g., *properties*) and does not have pointer variables.

In this paper, we present a new programming system, called PLDS, which augments our SpiceC [Feng et al. 2011] system with the ability to exploit parallelism in C++ programs that rely upon the use of dynamic linked data structures. In PLDS programmers do not have to write any code to handle misspeculation check and recovery. Since dynamic linked data structures are connected in a loosely-coupled manner, we improve the efficiency of speculation execution based on them with a novel speculation mechanism - *conditional speculation*. PLDS supports data partitioning and then distribution of computation across threads based upon the partitioning. It improves data locality for parallel programs and reduces misspeculation rate when speculative parallelism is used. To make programming easier, PLDS provides a set of easy-to-use compiler directives for data partitioning and computation distribution across threads based on the partitioning. Developers can choose amongst variety of data partitionings and distribute computation across threads in a partitioning sensitive fashion. Developers do not write new parallel programs but add PLDS pragmas into the text of an existing program. PLDS supports two major features - *multiple forms of parallelism* and *conditional speculation*, which none of the existing partitioning-based programming models support.

Multiple forms of parallelism. To make PLDS versatile, PLDS supports different types of parallelism – homogeneous parallelism and heterogeneous parallelism. Homogeneous parallelism is similar to Single Program Multiple Data (SPMD) parallelism. All threads execute the same code but operate on different data partitions. In heterogeneous parallelism, only the master thread starts to execute the code after entering a parallel region and all other threads are put in idle state. When a thread comes across a computation that requires data from another thread's partition, it assigns the computation to the corresponding thread.

Conditional speculation. Computations on dynamic linked data structures may need data from multiple threads' partitions. Such computations require the use of speculation to exploit dynamic parallelism. The cost of speculation may be very high since a computation may touch a large number of data objects. PLDS supports a *conditional speculation mechanism* that reduces the cost of speculation. It dynamically avoids the expense of speculation when threads happen to only touch their local partitions.

We present our design and implementation of PLDS in this paper. The core components of the PLDS prototype implementation consist of: a source-to-source translator and a user-level runtime library. The translator analyzes a loop and its PLDS directives and translates them into C/C++ code. The runtime library implements data partitioning and thread management. We evaluate our implementation of PLDS on an 8-core machine using ten benchmarks, of which six are parallelized using speculation. Our implementation achieves 1.3x–6.9x speedup for these benchmarks.

The remainder of the paper is organized as follows. Section 2 presents an overview of PLDS. Section 3 illustrates strategies for partitioning data objects belonging to a dynamically created linked data structure. In Section 4, we describe the programming interface and show how programs are written in PLDS. Section 4 also presents our conditional speculation mechanism. In Section 5, we describe the implementation of

PLDS. Section 6 shows the evaluation of PLDS. Section 7 discusses the related work and Section 8 concludes the paper.

2. PLDS OVERVIEW

PLDS exploits parallelism by distributing computations across threads that operate upon different data partitions created by dividing a dynamic linked data structure. The runtime execution flow of a PLDS program is as follows. A set of threads is created at the beginning of the program. Each thread is bound to a unique processor core. A master thread executes the sequential part of the program. When encountering a parallel region identified by the programmer, the master thread divides the data into multiple partitions via partitioning strategy selected by the programmer and maps each partition to one of the threads. Within a parallel region, each thread typically performs the computations that work on its assigned data partition and it either skips or redistributes the computations that work on other partitions. To support the above execution model, PLDS provides the following.

Partitioning Support. PLDS provides four data partitioning strategies to handle different data structures and computation patterns. METIS and HASH are two partitioning strategies for graph data structures with each computation working on one or more nodes. SYMM_SUBTREE and ASYMM_SUBTREE are two partitioning strategies for tree data structures with each computation exhibiting locality on a subtree. PLDS also allows the programmers to specify custom data partitionings.

Homogeneous Parallel Regions. In this form of parallelism, every thread executes the same code after entering the parallel region. Although all threads execute the same code, different threads perform computations on different data partitions. This parallelism is similar to the Single Program Multiple Data (SPMD) parallelism. Our programming model provides constructs for checking if the data required by a computation (i.e., an iteration of the loop) is located in the current thread's data partition. At runtime, a thread performs this check at an early stage of each computation. If the data required by a computation is located in the thread's partition, the thread continues executing the computation. Otherwise, the thread skips to the next computation. This execution model is often used for graph data structures with each computation starting from a different node.

Heterogeneous Parallel Regions. This form of parallelism is suitable for computations that always start from the same node (e.g., a search from the root of a tree). Homogeneous parallelism is not suitable here since the thread owning the starting node will end up performing all computations. After entering a heterogeneous parallel region, only the master thread starts to execute the code in the parallel region and all other threads are put in idle state. When the master thread comes across a computation that requires data from another thread's partition, it assigns the computation to the corresponding thread and continues to execute the code following the assigned computation. The thread that receives the computation then performs it in parallel with other threads. All threads are able to distribute computations to other threads based on the data required by the computations.

Speculative Parallelism. Sometimes, a computation may need to access and update the data from multiple partitions, where some partitions belong to other threads. For example, in a graph data structure, a computation may start from one node but end up with updating multiple nodes around the starting node. The thread that performs the computation may have to access the data from a partition assigned to another thread. In this case, it is possible that multiple parallel threads access the same data simultaneously. PLDS provides speculation support for this type of computation to re-

solve any data conflict between two threads. PLDS also supports a speculation mechanism called *conditional speculation*, which selectively applies speculation on computations to reduce the speculation cost.

In subsequent sections we present the partitioning strategies supported by PLDS and programming support given to the developer in form of pragmas.

3. PARTITIONING SUPPORT

Let us consider the strategies for partitioning data objects belonging to a dynamically created linked data structure. We provide programmers four partitioning strategies that can be used to handle commonly occurring scenarios involving graphs and trees used by a wide range of applications. These strategies include: METIS and HASH for graphs; and SYMM_SUBTREE and ASYMM_SUBTREE for trees (see Table I). We also provide the programmer with the ability to specify custom data partitioning strategies. In the remainder of this section we discuss the partitioning strategies in detail.

Table I. Summary of partitioning strategies.

Strategy	Struct.	Selection Criterion	Benefits
METIS	Graph	Spatial Locality Present	Locality[1]; Speculation[2]
HASH	Graph	No Spatial Locality	Locality[1]
ASYMM_SUBTREE	Tree	Searches Originating at Internal/Leaf Node Recursively Parallel Computation	Locality[1]; Speculation[2] Locality[1]
SYMM_SUBTREE	Tree	Searches Originating at Root Node	Locality[1]; Speculation[2]

[1]Improved cache locality. [2]Reduced speculation cost.

3.1. Graph Partitioning

Graphs are widely used in the design of algorithms. A graph data structure is defined as a set of data objects connected by edges. Each object is called a node or a vertex belonging to the graph. A graph may be directed or undirected. We divide the access patterns of graph data structures into two categories according to the presence or absence of *spatial locality* in the access patterns. *By spatial locality we mean that if a node is accessed by a computation, then it is likely that nodes adjacent to it in the graph will also be accessed during the same computation.* Next, we give examples that show that in the presence of spatial locality, METIS is a good partitioning strategy while in the absence of spatial locality HASH is a good strategy.

METIS. A computation on a graph data structure often requires accessing a set of nodes that are connected by edges. Often a computation begins by accessing one or more nodes in the graph. It then gradually involves more and more nodes that are adjacent to the beginning nodes. Thus, the access pattern exhibits spatial locality with respect to the graph structure. Figure 1 shows an example of the graph data structure where accesses exhibit spatial locality. A dark node indicates that the node is required by two different computations. Each computation requires accessing a connected subgraph and thus exhibits spatial locality with respect to the graph.

For this access pattern, if we randomly assign the computations to threads, we may incur many cache coherence misses. To improve the locality, we can group nodes that are near each other into partitions and assign each partition to a thread. Each thread can then perform computations that start from the nodes in its own partition. Since there is greater chance that each thread will access its own partition, the cache locality is enhanced. When threads are executed speculatively in parallel, this approach will result in low misspeculation rate since it is less likely that two parallel threads will access the same nodes.

Many algorithms have been proposed for partitioning graph data structures. We adopt the METIS algorithm, which is based on the state-of-the-art multilevel graph

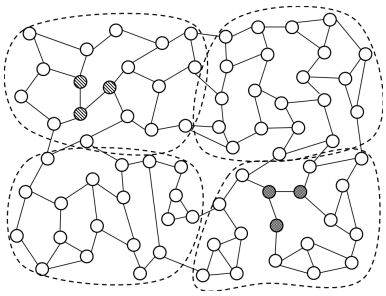


Fig. 1. Parallel computation in a graph with spatial locality.

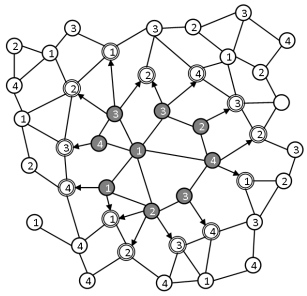


Fig. 2. Breadth first search - an example of parallel computation in a graph without spatial locality.

partitioning [Karypis and Kumar 1998]. The METIS produces very high quality partitionings for large graphs. As it operates with a reduced-size graph, it is extremely fast compared to traditional partitioning algorithms. Moreover, METIS has a parallel implementation, which is suitable for parallel programs.

In addition to partitioning an existing graph, we must also consider *repartitioning* as a result of node insertions. Since node insertions occur frequently for some applications (e.g., Delaunay Mesh Refinement [Hudson et al. 2007]), we require the repartitioning to be inexpensive. Instead of repartitioning the whole graph, we simply incorporate the new nodes into existing partitions. A new node is assigned to the partition, to which most of its neighbors belong. In this way, we minimize the number of edges that cross partition boundaries.

HASH. In some parallel applications, a computation block (e.g., a loop iteration) only accesses one node in the graph data structure. Thus this access pattern is without spatial locality. Figure 2 shows an example of parallel breadth first search (BFS) (see Figure 8 for pseudocode). The dark nodes are the nodes that have been visited. The double circles mark the nodes that need to be visited in parallel. Since each computation only accesses one node, grouping nearby nodes cannot improve the cache locality in this case. Besides, grouping nodes that are close to each other may even make the workload unbalanced since it is likely that the nodes that need to be processed in parallel are connected. For this type of access pattern, we just need to hash the nodes into partitions. We call this partitioning strategy as HASH. Each thread is assigned a partition constructed via hashing and the thread only processes the nodes in its own partition. This approach not only balances the workload but also improves cache locality if multiple BFSs need to be performed on the graph. In this scheme, partitioning and *repartitioning* are both performed via hashing.

3.2. Tree Partitioning

Tree data structures have been widely used to represent hierarchical structures. We propose two partitioning strategies for tree structures which are described next.

ASYMM_SUBTREE. This partitioning strategy supports two types of tree computations: computations that start from an internal/leaf node and recursive computations. First let us consider the computations that start from an internal/leaf node and then travel within a local subtree that contains the node. Figure 3 shows two examples of such computations – one computation begins at an internal node and the other at a leaf node but both computations only touch a subtree instead of the whole tree. This behavior is typical of searches on trees such as quadtrees and KD-trees. For a program that performs a large number of such computations at runtime, performing these com-

putations in parallel could greatly improve the performance. In this scenario we would like repeated computations on the same subtree to be performed by the same thread to achieve improved cache locality and also reduce the chance of two parallel threads accessing the same data. To achieve this we support a partitioning strategy that forms partitions from subtrees as illustrated by the partitions shown in Figure 3 by dotted lines. Since this strategy creates subtrees that are not of exact same size we name it ASYMM_SUBTREE. The resulting subtrees are assigned to different threads, which can operate on their respective partitions in parallel. A computation is performed by the thread that owns the starting node. For example, in Figure 3, the computation on the left side is performed by the thread owning the leftmost partition and the computation on the right side is performed by the thread owning the rightmost partition.

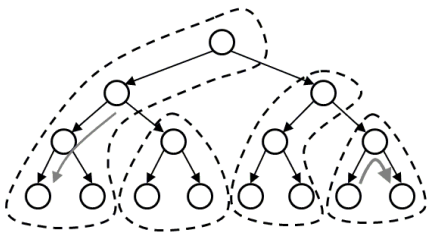


Fig. 3. Parallel computations starting at an internal/leaf node.

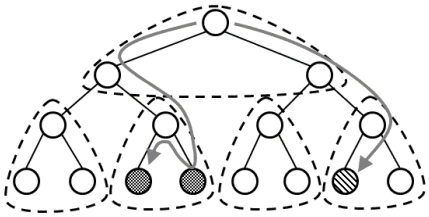


Fig. 4. Parallel computations starting from the root node.

Now let us consider the case of recursive parallel computations. In such computations, typically, the starting thread first assigns all of the child subtrees except one to other parallel threads and then continues to process the one left. The threads assigned the child trees repeat the above procedure until no more threads are available to assign part of the work. Such recursive parallel code is invoked multiple times in many programs we have examined. If we randomly assign threads to subtrees for each invocation of the parallel code, we will lose the opportunities of reusing the data in the cache across multiple invocations. Even if the subtrees are assigned to threads in the same fixed order during each invocation, the mapping of partitions to threads may vary across different invocations. To improve data reuse, we can partition the tree as shown in Figure 3, and at runtime, require each thread to process the same subtree.

```
root.partition ← 0;
partitions ← 1;
nodequeue ← {root};
while ( nodequeue ≠ φ ) {
    nodequeue.pop(n);
    foreach child ∈ n.childset do {
        nodequeue.push(child);
        if (child is leftest OR partitions = total_partitions)
            child.partition ← n.partition;
        else {
            child.partition ← partitions;
            partitions ← partitions + 1;
        }
    }
}
```

Fig. 5. Asymmetric tree partitioning algorithm.

Figure 5 shows the algorithm for asymmetric tree partitioning shown in Figure 3. The algorithm starts with creating a new partition for the root. When the number of

partitions is smaller than the limit, the algorithm assigns a node's leftmost child to the same partition and creates new partitions for all other children. Once the number of partitions reaches a limit, no new partitions are created. Each of the remaining nodes is assigned to its parent's partition. The *repartitioning* strategy allows for leaf node insertion and it assigns a new leaf node to its parent's partition.

SYMM_SUBTREE. Most tree algorithms always start at the root node. Figure 4 shows two examples of tree computations starting at the root node. The right arrow indicates a tree computation that goes from the root to a leaf node and updates the value at the leaf node (e.g., binary insertion). The left arrow indicates a tree computation that goes from the root, searches a subtree, and finally updates the values at several nodes (e.g., during local tree balancing). A tree-based program typically begins many computations starting from the root node at runtime. Performing these computations in parallel can greatly improve its performance. For such computations, if each thread only processes the operations on a certain subtree, then we can reduce the misspeculation rate or even do without speculation (e.g., during binary search). This also improves the cache locality. Since every computation starts from the root node, all computations will be started in the thread that owns the root node. We will get poor performance if we use ASYMM_SUBTREE partitioning algorithm in this case. For example, under the partitioning of Figure 3, all computations will start in the thread owning the leftmost partition. If a computation goes right, the thread will assign it to another thread. However, if a computation goes all the way left, all subsequent computations will be blocked until it is completed. Suppose a computation has equal chance of going left and right, around 1/4 of the computations will block subsequent computations. This will greatly reduce the parallelism. Moreover, if a computation goes all the way to the right, it will go through three threads and incur communication overhead.

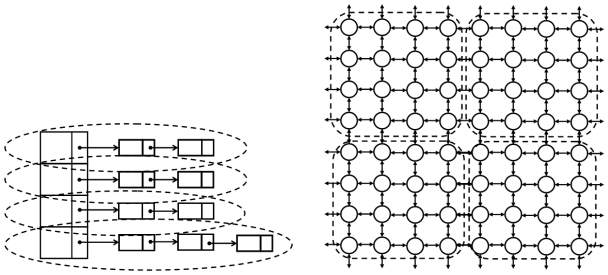
To address the above problem we propose a strategy that partitions the tree into symmetric subtrees as shown in Figure 4 - we name it SYMM_SUBTREE as it partitions the tree in symmetric subtrees. A *root partition* is created to contain the nodes at the top few levels of the tree while each *subtree partition* contains a subtree under the root partition. At runtime, each thread is assigned a subtree partition. The root partition can be assigned either to the master thread or shared by all threads. Figure 6 presents the symmetric tree partitioning algorithm. The algorithm starts with assigning nodes to the root partition from the top of the tree until reaching a level, at which the number of nodes is greater that the number of desired partitions. It then evenly distributes those nodes among the partitions. We do not need to assign the remaining nodes to the partitions since the searches always go from the top to the bottom.

3.3. Programmer Specified Partitioning

Programmers often have the knowledge of the shape of the data structures that can be exploited by them during partitioning. To support this scenario we also allow programmers to specify their own partitioning strategies. Figure 7 shows two partitioning examples that can be easily specified by programmers. Figure 7(a) shows a hash table that consists of an array of linked lists. The programmer can easily group the linked lists by hashing the array indices into partition IDs. At runtime, each thread only processes hash operations in its own partition. This partitioning will improve the cache locality and avoid data contention. Figure 7(b) shows an example of a grid computation. The computation on each node requires the data from its neighbors. The programmer can easily group nodes near each other into the same partition given the position of each node. Although both examples can be partitioned using METIS, programmer specified partitionings as shown are even superior.

```
nodeset ← root;
while ( |nodeset| < total_partitions ) {
    childset ← φ;
    foreach node ∈ nodeset do
        childset ← childset + node.childset;
    nodeset ← childset;
}
size ← ⌈total_partitions / |nodeset|⌉;
partitions ← 0;
while ( partitions < total_partitions ) {
    for i ← 1 to size do
        if ( nodeset ≠ φ ) {
            nodeset.pop(n);
            n.partition ← partitions;
        }
    partitions ← partitions + 1;
}
```

Fig. 6. Symmetric tree partitioning algorithm.



(a) Hash table (b) Grid computation

Fig. 7. Example of manual partitioning.

4. PROGRAMMING SUPPORT

This section presents our programming support for using partitioning based approach in programming applications. This support is in the form of OpenMP-like pragmas. We divide the presentation of these pragmas according to the types of parallelism they support: *homogeneous*, *heterogeneous*, and *speculative*. We also present examples to illustrate the use of these pragmas in variety of application scenarios.

To enable partitioning of a linked data structure, the partitioning algorithm must be able to identify the data structure, i.e, its nodes and edges. To allow this our approach requires that the data structures that need to be partitioned inherit a base class, called `BaseNode`. The `BaseNode` class defines two interfaces (i.e., virtual functions in C++) – `GetNeighborNum` and `GetNeighbor` which developers must implement. The first interface returns the total number of neighbors and the second returns a specified neighbor. Our partitioning library can acquire the graph information by calling these two functions. Besides, the implementation of the `BaseNode` should automatically maintain a list of nodes that have not been partitioned. The `BaseNode` class should also provide two member functions – `GetPartition` and `SetPartition`. These two functions are used to get and set the partition to which a data object belongs.

4.1. Non-speculative Homogeneous Parallelism

```
#pragma homo partition (DataType,PartitionType)
#pragma inpartition (partition)
```

The first pragma is used to mark the code region, such as a loop, that is to be executed in parallel and where partitioning is to be applied. The `homo` clause indicates that

homogeneous parallelism is to be used and the `partition` clause indicates the type of partitioning where the parameter `DataType` gives the type of the data structures to be partitioned and the parameter `PartitionType` identifies the partitioning strategy to be used. The second pragma is used to check in which thread the subsequent computation is to be performed. The pragma checks if the partition specified by the parameter belongs to the current thread. If so, the current thread continues the subsequent computation. Otherwise, the current thread skips the rest of the computation and goes to the next computation. The `partition` parameter can be given in the form of constant, variable, or return value of a function. Next we illustrate the use of above pragmas through examples.

```
read(graph);
vertexset.add(source);
while( !vertexset.empty() ) {
    nextset.clear();
    #pragma homo partition(Vertex, HASH)
    for(int i=0; i<vertexset.size(); i++) {
        v = vertexset[i];
        #pragma inpartition(v.GetPartition())
        if ( !v.visited) {
            v.visited = true;
            process(v);
            nextset.lockedAdd(v.adjacentset());
        }
    }
    vertexset = nextset;
}
```

Fig. 8. Pseudocode for parallel BFS with graph partitioning.

Parallel loop with graph data structures. Figure 8 shows the pseudocode of a parallel implementation of Breadth First Search (BFS). The vertexset and nextset are containers that store unique vertices. The starting node is placed in the vertexset at the beginning. In the parallel loop, each iteration processes one node in the vertexset and adds pointers to its neighbors to the nextset. The program does not examine the visited bit of a neighbor when adding it to the nextset to avoid excessive cache coherence misses. After the loop has completed, the program moves the nextset to the vertexset and repeats the loop. The program ends when all nodes have been visited.

We can use HASH partitioning for the graph since each computation of the parallel BFS just accesses one node. Partitioning can improve the data reuse if the BFS is called multiple times. We use two pragmas to realize the partitioning-based parallelism in this case. The first pragma declares that the code region must be executed in the form of homogeneous parallelism and the graph must be partitioned using the HASH strategy. Partitioning will only be performed once by the program since the unpartitioned node list will be empty after the first partitioning. The second pragma checks if the vertex belongs to the current thread's partition. If so, the current thread continues the iteration. Otherwise, the current thread skips to the next iteration. The function `GetPartition` returns the vertex's partition that is computed by the first pragma. At runtime, the master thread executes the program until reaching the first pragma. After data partitioning and mapping, all threads execute the code region in parallel. Each thread performs the computations on the vertices in its own partition.

Programmer specified partitioning. Our pragmas also support programmer specified partitioning. Figure 9 shows an example of parallel hash table lookup. Before entering the loop, the programmer uses the function `SetPartition` to manually assign each entry in the hash table to a partition. The pragma *homo* indicates that the

loop will be carried out by all threads. Each thread performs the hash function for every work in the worklist but only does the insertion for the work that is hashed into its partition. If we do not partition the hash table and randomly distribute works among threads, we will need speculation since two threads may insert nodes into the same part of the hash table. Through partitioning, we eliminate the need for speculation.

```
for(index=0; index < table.size(); index ++)
    table[index].SetPartition(index % totalthreads());
#pragma homo partition(NONE)
foreach w in worklist do {
    index = hash(w);
    #pragma inpartition(table[index].GetPartition()) {
        if ( table[index].lookup(w) == NULL )
            table[index].insert(w);
    }
}
```

Fig. 9. Parallel hash table lookup with partitioning.

4.2. Non-speculative Heterogeneous Parallelism

```
#pragma hetero partition (DataType,PartitionType)
#pragma distribute inpartition (partition)
#pragma join
```

The first pragma is used to mark the code region such as a loop that is to be executed in parallel and where partitioning is to be applied. The `hetero` clause indicates that heterogeneous parallelism is to be used and the `partition` clause indicates the type of partitioning to be used. The second pragma is used to check in which thread the subsequent computation should be performed. The `distribute` clause indicates that the pragma will try to assign the subsequent computation to another thread to which the partition belongs so that the subsequent computation can be performed in parallel. If the partition belongs to the current thread, the subsequent computation is skipped. Finally, the third pragma, `#pragma join`, is used for synchronization between threads. The current thread will sleep until the thread owning the given partition finishes its computation. If the current thread owns the given partition, the pragma does nothing. Next we illustrate the use of above pragmas through examples.

```
root=tree.root();
#pragma hetero partition(Node, SYMM_SUBTREE)
for(int i=0; i<worklist.size(); i++) {
    value = worklist[i];
    node = root;
    prev = NULL;
    while ( node != NULL &&
          node.GetPartition() == currentPartition() ) {
        prev = node;
        node = node.search(value);
    }
    if (node == NULL)
        insert(prev, value);
    else {
        #pragma distribute inpartition(node.GetPartition()) {
            node.searchAndInsert(value);
} } }
```

Fig. 10. Pseudocode for parallel binary search with tree partitioning.

Parallel loop with tree data structures. Figure 10 shows the pseudocode for parallel binary search and insertion. For each work item in the worklist, the program always starts search from the root node to a leaf node. If we do not partition the tree, the search procedure requires speculation since multiple searches may go through the same path. To realize partitioning-based parallelism for this program, we need to modify the program a little as shown in highlight. The first pragma partitions the tree using the SYMM_SUBTREE partitioning and maps the partitions to the processor cores. Since we do not know in which subtree each work in the worklist will be located before performing the search, we mark the loop heterogeneous so it is only executed by the master thread. In each iteration, the master thread performs searches until it reaches a node that is not in its partition. Then the second pragma distributes (assigns) the continuation of the search to a parallel thread corresponding to the node's partition. By partitioning the tree in this manner, the searches that are located in the same subtree will be performed by the same thread. Therefore, we improve the performance of the program without requiring the use of speculation.

```
function update(node) {
    pre_process(node);
    foreach child in node.children
        #pragma distribute inpartition(child.GetPartition())
            update(child);
    foreach child in node.children
        #pragma inpartition(child.GetPartition())
            update(child);
    foreach child in node.children
        #pragma join(child.GetPartition())
    post_process(node);
}

...
// in main function
#pragma hetero partition(Node, ASYMM_SUBTREE)
update(root);
...
```

Fig. 11. Pseudocode for parallel recursion computation with tree partitioning.

Recursive computation with tree data structures. Figure 11 shows a simple example of recursion parallelism. The recursive function is called for multiple time at runtime. To improve the data reuse, we can partition the tree using the ASYMM_SUBTREE strategy. We use the pragma to partition the tree in the main function. In the recursive function, we check the partition of each child of the node. If a child belongs to another thread's partition, the update for that child is assigned to that thread. After assigning all the children that are not in the current thread's partition, the current thread continues to update the rest children. After the current thread completes its work, it waits for other threads to finish their work using `join` and then performs the `post_process` work. Since the same thread always updates the same child, the cache performance of the program is improved by our approach.

4.3. Speculative Parallelism

For parallel programs where each computation may touch data across multiple partitions simultaneously, we provide support for speculation via the following pragmas.

```
#pragma [conditional] speculate
#pragma repartition
#pragma touchpostpone(partition)
```

The first pragma is used to mark the code region that requires speculative execution to enforce atomicity. When the `conditional` clause is not specified, standard speculation mechanism is used, i.e., speculation is used for every execution of the code region. When the `conditional` clause is specified, the conditional speculation mechanism is used for the region where speculation is only used for computations that dynamically are determined to require access to multiple threads' partitions. The second pragma is used to incrementally assign partitions to newly created nodes. Although we illustrate this pragma in context of speculation, it can also be used in non-speculative situations as needed. The third pragma is used to assist in conditional speculation. It postpones the execution of current computation if the given partition is not a local partition. The postponed computation is later executed using standard (unconditional) speculation mechanism. We illustrate the usage of above pragmas using an example.

```
read(mesh);
worklist.add(mesh.getBads());
while( !worklist.empty() ) {
    todolist.clear();
    #pragma homo partition(Triangle, METIS)
    for(int i=0;i<worklist.size();i++) {
        #pragma inpartition(worklist[i].GetPartition())
        #pragma speculate {
            Cavity c = new Cavity(worklist[i]);
            c.build(mesh);
            c.retriangulate(mesh);
            mesh.update(c);
            #pragma repartition
        }
        todolist.lockedAdd(c.newBads());
    }
    worklist = todolist;
}
```

Fig. 12. Parallel Delaunay Mesh Refinement with graph partitioning.

Figure 12 shows the pseudocode of a parallel implementation of Delaunay Mesh Refinement [Hudson et al. 2007]. The program is designed to refine bad triangles (i.e., triangles whose circumradius-to-shortest edge ratios are larger than some bound) in a mesh. It takes a triangular mesh as input. Each iteration of the parallel loop first searches the triangles around one bad triangle to form a cavity, then retriangulates the cavity, and finally updates the mesh. If the procedure generates a new bad triangle, it is placed in the *todolist*. The parallel loop requires speculation since two iterations may access the same triangles. After the parallel loop is completed, the program moves the todolist to the worklist and repeat the parallel loop. The program ends when there is no bad triangle in the mesh. Since the bad triangles can be processed in any order, there is no need to enforce any order in the parallel implementation.

The data accesses of the Delaunay Mesh Refinement algorithm exhibit spatial locality since if a triangle is accessed in a computation its neighbors are likely accessed in the same computation. We can improve the program's performance by grouping nearby nodes into partitions. We realize the partitioning-based parallelism by using three pragmas. The first pragma declares the code region that will be executed by all threads, partitions the mesh (a graph where each node is a triangle) with the METIS algorithm, and maps the partitions to the threads. The partitioning is only done once in the program since the list of unpartitioned nodes will be empty in the subsequent execution. The second pragma checks if the computation should be done in the current thread. If the computation should be done in another thread, the current thread moves

on to the next iteration. The function `GetPartition` returns the bad triangle's partition that is computed by the first pragma. The third pragma incrementally updates the partitioning after new triangles are added into the mesh. At runtime, the master thread executes the program until reaching the first pragma. After data partitioning and mapping, all threads execute the parallel code region in parallel as indicated by the `homo` clause. Each thread only performs the computations that begin from its own partition and skips the rest computations. After retriangulating a cavity, a thread incrementally partitions the newly created triangles using the `repartition` pragma.

Conditional speculation. Although data partitioning can reduce the misspeculation rate, the high overhead of speculation can still hurt the parallel performance. In fact, under partitioning-based parallelism, most computations are likely to touch their local partitions as long as each partition is large enough. When all threads are performing computations that only touch their own partitions, the speculation is not needed. Motivated by this observation, we provide support for partitioning-based *conditional speculation*. In this approach, by default a thread does not start speculation at the beginning of each computation. If a computation only touches data in its local partition, then it succeeds without speculation. If the thread detects that a computation will touch other partitions, the thread postpones its execution for later. When all threads complete the computations on their local partitions, they use speculation in redoing all postponed computations. The conditional use of speculation cuts down the percentage of computations executed under the speculation though it causes extra overhead for the computations involving multiple partitions. Therefore, as long as most computations only touch local partitions, the overall cost of speculation is reduced.

As we mentioned before, for the computations that require a subset of nodes in the graph, usually start with collecting the nearby nodes around the starting node. Since the collection procedure usually does not write to any global variable such as the graph data structure, it can be executed without speculation. In the collection procedure, a computation decides which nodes it will use later. Therefore, during the collection procedure, we can determine which partitions a computation will touch. If a computation will touch a remote partition, the thread postpones the computation, saving the loop index in a queue for later, and continues to the next computation. These saved computations will be executed using speculation after all other computations are done.

```
frontier.add(center);
foreach node in frontier do {
    foreach neighbor in node.getNeighbors() do {
        #pragma touchpostpone(neighbor.GetPartition())
        if ( neighbor is part of the cavity ) {
            cavity.add(neighbor);
            frontier.add(neighbor);
        } else {
            border.add(neighbor);
} } }
```

Fig. 13. Pseudocode for building cavity in Delaunay Mesh Refinement.

For example, in Figure 12, the triangles that will be touched are determined in function `build`. Figure 13 shows the pseudocode of the `build` function. The function collects the triangles that are in or around the cavity. The function only writes to the variables local to the threads. Therefore, the function can be executed without speculation. To realize adaptive speculation, we add one pragma (highlighted) into the function. The pragma first checks the partition of variable `neighbor`. If `neighbor` is in the local partition, the computation continues. If it is in a remote partition, the pragma saves the loop index of the computation (i.e., variable `i` on line 6 in Figure 12) in a queue

and skips to the next computation. In the parallel loop shown in Figure 12, developers just need to wrap the speculative code using `#pragma conditional speculate` instead of `#pragma speculate`. No other change needs to be made. At runtime, in the main parallel loop of Delaunay Mesh Refinement (as shown in Figure 12), the `#pragma conditional speculate` does nothing by default. Each thread checks the partition of each triangle at the pragma `touchpostpone`. After all threads complete the parallel loop, they redo the computations for the saved indices. During this redo procedure, the `#pragma conditional speculate` starts and ends speculative execution normally.

For the correctness purpose, we do not allow write to any global variable before the collection procedure is finished. Our compiler can detect whether there is a write to global variable along the paths from "#pragma speculate" to "#pragma touchpostpone". If there is such a write, our compiler will give a warning to the programmer.

5. IMPLEMENTATION OF PLDS

This section briefly describes the implementation of PLDS. The core components of the PLDS prototype implementation consist of: a source-to-source translator and a user-level runtime library. The translator analyzes the loop and PLDS directives and translates them into C/C++ code. The analysis is done by extending ROSE [Quinlan 2000], which is an open source compiler infrastructure to build source-to-source code translator. The runtime library implements data partitioning and thread management.

Pragmas. The `inpartition` and `touchpostpone` pragmas can cause a thread to skip the rest of the current computation and jump to the next computation. We allow them to be placed inside a function called in the parallel region. They are implemented using the exception support in C++. When these pragmas decide to skip the current computation, they throw an exception. The parallel code region is wrapped with an exception handler. Once it captures an exception from those pragmas, it jumps to the next computation. The *speculation pragmas* are implemented using the Intel STM Compiler [Intel 2010], which implements Software Transactional Memory (STM) for C/C++ in a compiler. With this compiler, transforming a piece of code into a transaction requires very few changes to the code.

Thread management. We use the *pthread* library for creating and managing threads. Threads' context is stored in thread local storage. All working threads are created at the beginning of the program. Multiple executions of a parallel region reuse the same threads making the cost of thread creation to be amortized. We use busy-waiting algorithms to implement the synchronizations between threads. The busy-waiting algorithms achieve low wake-up latency and hence yield good performance [Mellor-Crummey and Scott 1991].

Threads-to-cores mapping. In a parallel region using the *speculation* pragma, each computation may access multiple adjacent partitions. In modern multicore processors, the caches are organized in a hierarchy that causes closer cores to share more efficiently. To improve the utilization of the cache hierarchy, we need to place adjacent partitions to close cores so that accessing adjacent partitions will be faster. We use the cache topology aware mapping algorithm proposed in [Kandemir et al. 2010] to map the threads to the processor cores for parallel region using speculation.

6. EVALUATION

This section evaluates our prototype implementations of PLDS. The experiments are conducted on an 8-core DELL PowerEdge T605 machine. Table II lists the details of the machine. The machine runs *CentOS v5.5*.

6.1. Benchmarks

We use 10 benchmarks in the experiments. Two out of ten benchmarks are real applications. The rest eight benchmarks are from three benchmark suites – Lonestar [Kulka-

Table II. Machine details.

Processors	2×4-core AMD Opteron processors (2.0GHz)
L1 cache	Private, 64KB for each core
L2 cache	Private, 512KB for each core
L3 cache	Shared among 4 cores, 2048KB
Memory	8GB RAM

Table III. Benchmark details. From left to right: benchmark name; function where the parallel region is located and type of parallelism (homogeneous and heterogeneous); lines of code in the function and number of pragmas introduced; type of data structure used in the benchmark; partitioning strategy employed; whether speculation is used and whether conditional speculation is used.

Benchmark	Function (Par. Type)	LOC + #Pragmas	Data Structures	Partitioning	Spec.? - Cond.?
Delaunay-Refinement	main (homo.)	108 + 5	graph	METIS	yes - yes
Boykov-Kolmogorov	main (homo.)	93 + 5	graph	METIS	yes - yes
	main (homo.)	93 + 3	graph	HASH	no - no
Barnes-Hut	main (homo.)	107 + 2	graph	METIS	no - no
Agglomerative-Clustering	clustering (homo.)	61 + 4	tree	ASYMM_SUBTREE	yes - yes
Voronoi	build_delaunay (heter.)	106 + 3	tree	ASYMM_SUBTREE	no - no
TreeAdd	TreeAdd (heter.)	23 + 3	tree	ASYMM_SUBTREE	no - no
AVL	main (heter.)	87 + 4	tree	SYMM_SUBTREE	yes - no
ITI	batch_train (heter.)	60 + 4	tree	SYMM_SUBTREE	yes - no
Hash	main (homo.)	31 + 2	hash table	programmer specified	no - no
Coloring	coloring (homo.)	32 + 5	graph	programmer specified	yes - yes

rni et al. 2009], Olden [Carlisle and Rogers 1995], and Shootout. All benchmarks either already use pointer-linked data structures or were ported to make use of them. For all benchmarks, the order in which the computations are performed does not affect the correctness of the output though different orders may yield different outputs. Table III shows the details of the benchmarks.

The benchmarks `Delaunay Refinement` and `Agglomerative Clustering` are from the Lonestar benchmark suite [Kulkarni et al. 2009] and originally written in JAVA. We ported them into C++. The original `Delaunay Refinement` uses a few hash table-based containers to store the graph. Therefore, even if two computations require different parts of the graph, they may share some data in the containers. We rewrote the program using pointer-linked data structures to solve this problem. `Agglomerative Clustering` is a hierarchical clustering algorithm. It uses a KD-tree for nearest neighbor search. We reorganized the code so that a node collection procedure is conducted at an early stage of each computation. `Barnes-Hut` is an implementation of the Barnes-Hut n-body algorithm [Barnes and Hut 1986] that simulates the gravitational forces in a galactic cluster. It uses an octree for storing nodes in the graph. We parallelized the force-computation step in the main loop. The benchmarks `Voronoi` and `TreeAdd` are from the Olden benchmark suite [Carlisle and Rogers 1995]. `Voronoi` implements a recursive Delaunay Refinement algorithm. `TreeAdd` calculates the sum of values in a balanced B-tree. Both benchmarks are parallelized using recursion parallelism. In the experiments, the sum calculation was repeated for ten times during the execution of `TreeAdd`. `Boykov-Kolmogorov` is a maxflow algorithm used for image segmentation. We parallelized two different loops in this program using different partitionings – METIS and HASH, respectively. For HASH-based parallelism, although there is no data contention between threads on the graph structure, two threads may share other global variables. We use mutex to serialize the accesses to these variables. `AVL`

Table IV. Comparison of misspeculation rates with 8 threads. From left to right: benchmark name, misspeculation rate without partitioning, and misspeculation rate with partitioning.

Benchmark	w/o par.	w/ par.
DelaunayRefinement	90.42%	0.07%
Boykov-Kolmogorov	20%	0.09%
AgglomerativeClustering	3.21%	0.25%
AVL	21.84%	3.19%
ITI	16.58%	0.73%
Coloring	6.15%	0.17%

is a self-balancing binary tree algorithm designed for addressing the issue of deletions. During execution, balancing usually takes place locally. We parallelized it using SYMM_SUBTREE partitioning. `ITI` [Utgoff et al. 1997] is a real application that constructs decision tree automatically from labeled examples. We speculatively parallelized the `batch_train` function. The `Hash` benchmark is from the Shootout benchmark suite. The main loop performs a large number of hash table lookups and insertions. The hash table can be easily partitioned by programmers. `Coloring` is an implementation of the scalable graph coloring algorithm in [Boman et al. 2005]. The original program does not use graph partitioning or speculative parallelism. In the original program, threads communicate node information with each other. In the experiments, we parallelized the program using user-specified data partitioning and speculative parallelism.

6.2. Performance

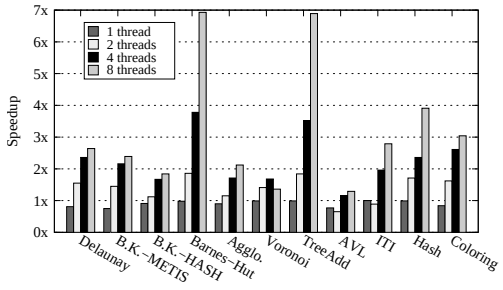


Fig. 14. Speedup by PLDS.

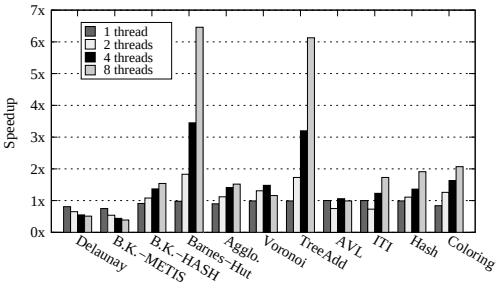


Fig. 15. Speedup without partitioning.

Figure 14 shows the speedup achieved by PLDS using different number of threads. The original sequential versions of these benchmarks are used as baseline. For every benchmark except `Voronoi`, the performance of the benchmark improves with the increase of thread number. The performance of `Voronoi` decreases at 8 threads due to the high communication overhead (as shown in Figure 17). `AVL` and `ITI` gets slowdown with two threads since it employs a master/worker execution model similar to the example given in Figure 10. With two threads, they have only one worker thread to perform computations. Figure 15 shows the speedup without partitioning, where computations are assigned to threads in a round-robin manner. *As we can see, the speedups achieved by PLDS are much higher than those without partitioning.* Without partitioning, parallelization causes slowdown for three benchmarks, for all of which PLDS achieves speedup.

Table IV compares the misspeculation rate with and without partitioning. *With partitioning, the misspeculation rates of all six benchmarks are greatly reduced.* The misspeculation rates of five out of six benchmarks almost decrease to zero. `Delaunay`

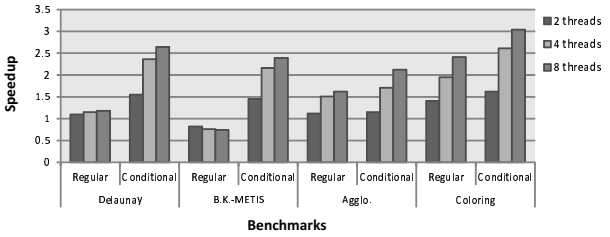


Fig. 16. Regular speculation vs. Conditional speculation

Table V. Postpone rate.

Benchmark	2 threads	4 threads	8 threads
DelaunayRefinement	3.5%	6.9%	13.1%
Boykov-Kolmogorov	4.6%	7.6%	15.4%
AgglomerativeClustering	1.7%	3.2%	6.0%
Coloring	2.1%	3.9%	8.7%

`Refinement` has the highest misspeculation rate when partitioning is not used. This is because close triangles are processed consecutively in the original program. With partitioning, most misspeculations of `Delaunay Refinement` are eliminated.

6.3. Effectiveness of Conditional Speculation

Figure 16 shows a performance comparison between regular speculation and conditional speculation on the four benchmarks – `Delaunay Refinement`, `Boykov-Kolmogorov`, `Agglomerative Clustering`, and `Coloring`. Without conditional speculation, parallelization causes slowdown for `Boykov-Kolmogorov` due to its high speculation overhead (as shown in Figure 17). Conditional speculation improves performance for all four benchmarks. *With 8 threads, conditional speculation improves the performance for the four benchmarks by 2.23x, 3.91x, 1.31x, and 1.46x over regular speculation, respectively.*

Table V shows the percentages of postponed computations for the four benchmarks. *For all four benchmarks, only less than one-sixth of computations requires speculation since most computations only access local data partitions.* The postpone rates increase as the number of threads is increased. This is because the chance of accessing multiple threads' partitions goes up as the size of each partition decreases. `Delaunay Refinement` and `Boykov-Kolmogorov` have similar postpone rates. The postpone rates of `Agglomerative Clustering` and `Coloring` are a little lower. However, due to the high cost of the node collection procedures in `Agglomerative Clustering`, its speedup is lower than that of the other two.

6.4. Overhead

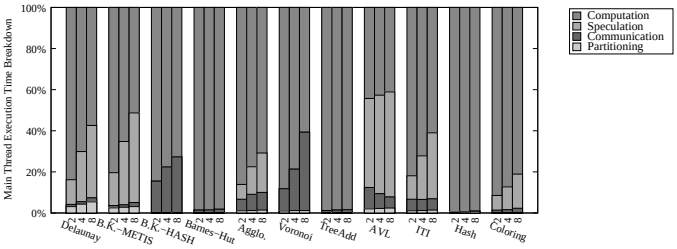


Fig. 17. Time breakdown of the master thread.

Figure 17 shows the time breakdown for the master thread with different numbers of threads. We divide the time into four categories: *computation, speculation, communication and partitioning*. Among the four categories, partitioning spends the least time. For the four benchmarks that use conditional speculation, their speculation overhead rises with the increase of thread number. This is mainly due to the increasing postpone rate as shown in Table V. Among all the benchmarks, `AVL` has the highest overhead. Its master thread spends half of the time on speculation since it uses regular speculation. `B.K.-HASH` and `Voronoi` have higher communication overhead which hurts their performance. Three benchmarks, Barnes-Hut, `Hash`, and `TreeAdd`, have very low overhead. Partitioning eliminates the speculation cost for `Hash`.

7. RELATED WORK

Data partitioning has been mostly explored for array-based data-parallel programs on distributed-memory computers [Kennedy and Allen 2001; Rogers and Pingali 1989]. Partitioned Global Address Space (PGAS) is a parallel programming model that explores data distribution for array-based data-parallel programs. It includes Unified Parallel C [Consortium 2005], Co-Array Fortran [Dotsenko et al. 2004], Titanium [Hilfinger et al. 2005], Chapel [Chamberlain et al. 2007], and X10 [Charles et al. 2005]. They all use SPMD programming style and some of them allow programmers to control data distribution. However, since these works focus on array-based data parallel programs, they do not need to consider dynamic computation partitioning and data contention between threads. High Performance Fortran was extended for dynamic data distribution to parallelize array-based unstructured computations [Müller and Rühl 1995]. Several works [wei Liao et al. 2006; Gordon et al. 2006] have focused on data and task partitioning for stream programs.

The Galois system has been extended to use data partitioning for optimizing worklist-based parallelism [Kulkarni et al. 2008]. PLDS is different from Galois in the following aspects. (1) Galois is not an automatic system while PLDS is. Galois is a runtime library and does not have compiler support. PLDS has both runtime library and compiler support. To use Galois, programmers need to use Galois-provided data strucutres when writing their programs. PLDS does not require this. (2) Galois does not provide full support for speculative parallelization since it does not support rollback. In Galois-provided data strucutres, locks are used to avoid conflicting accesses. However, when a lock cannot be acquired, a Galois thread only releases all acquired locks (in case of deadlock) and jumps back to the beginning of the computation. It does not rollback any performed operation due to the lack of compiler support. Our compiler translates code using transactional memory. Therefore, we support rollback. (3) We introduced partition-based heterogeneous parallelism, which Galois does not have. The function for parallel loop execution (i.e. Galois::for_each) provided by Galois only distributes iterations in a homogeneous way. Besides, Galois does not provide any function for synchronization between threads. (4) We proposed the conditional speculation scheme, with which most computation in a speculative parallel region can be done speculation-free. The Galois system does not support conditional speculation. As shown in our experiments, conditional speculation improves the performance by 2.48x on average over regular speculation.

Thread level speculation (TLS) techniques have been proposed to explore more parallelization opportunities for sequential programs. Krishnan et al. [Krishnan and Torrellas 1999] proposed an architecture for hardware-based TLS. Recently, many works have focused on software-based TLS. Behavior oriented parallelization (BOP) [Ding et al. 2007; Kelsey et al. 2009] is a process-based speculation technique. Copy or Discard (CorD) [Tian et al. 2008; 2010] is a thread-based speculation technique. All these TLS techniques are based on state separation. In other words, speculative computa-

tions are performed in a separate memory space. The results are not committed to the non-speculative space until the speculation succeeds. BOP [Ding et al. 2007; Kelsey et al. 2009] improves the performance of data copying between spaces with the help of OS. CorD was extended to support a set of compiler optimizations for copying dynamic data structures between spaces [Tian et al. 2010].

Partitioning has been explored for individual applications. Lin et al. [Lin et al. 2007] proposed to partition the variance of the noise-whitened encoding matrix for parallelizing Magnetic Resonance Imaging (MRI) reconstruction. Zeng et al. [Zeng et al. 1998] improve parallel simulation of large-scale wireless networks by partitioning the network to achieve load balance. Scott et al. [Scott et al. 2007] proposed to use partitioning for parallelizing Delaunay triangulation. In comparison with these works, our programming model is designed for general-purpose parallelization.

Many programming models have been proposed for writing parallel programs. OpenMP [Dagum and Menon 1998] is widely used model for parallelizing sequential programs on shared-memory systems. It expresses non-speculative parallelism using compiler directives. Threading Building Blocks (TBB) [Reinders 2007] is an Intel-designed model for parallelizing sequential programs. Instead of using compiler directives, TBB provides developers with a set of threadsafe containers to wrap their codes. Messaging Passing Interface (MPI) [Gropp et al. 1994] is a SPMD programming model. Using MPI, developers need to write explicitly parallel programs, including manually distributing workloads and handling communications and synchronization. *None of these programming models provide support for data partitioning based computation distribution and speculative parallelism which we support*. Our previous work on the SpiceC [Feng et al. 2011] provides a parallel programming model that supports multiple forms of parallelisms, including DOALL, DOACROSS, and pipelining. It also provides support for speculative parallelism and manycore processors. Through the addition of PLDS we are able to handle dynamic linked data structures in SpiceC.

8. CONCLUSION

In this paper, we present a programming model – PLDS for writing efficient parallel programs with pointer-linked data structures. PLDS provides support for data partitioning and conditional speculation which improves cache locality and reduces misspeculation rate and speculation cost. PLDS offers easy-to-use OpenMP-like programming constructs and supports two forms of parallelism which together cover a wide range of applications. Our experiments show that PLDS achieves 1.3x–6.9x speedups for ten benchmarks on an 8-core machine.

REFERENCES

ANDERSON, J. M. AND LAM, M. S. 1993. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI*.

BARNES, J. AND HUT, P. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature 324,* 4, 446–559.

BOMAN, E. B., BOZDAĞ, D., CATALYUREK, U., GEBREMEDHIN, A. H., AND MANNE, F. 2005. A scalable parallel graph coloring algorithm for distributed memory computers. In *EURO-PAR*. 241–251.

CARLISLE, M. C. AND ROGERS, A. 1995. Software caching and computation migration in olden. In *PPoPP*. 29–38.

CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. 2007. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications 21,* 3, 291–312.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*. 519–538.

CONSORTIUM, U. 2005. Upc language specifications, v1.2. *Lawrence Berkeley National Lab Tech Report LBNL-59208*.

DAGUM, L. AND MENON, R. 1998. Openmp: An industry-standard api for shared-memory programming. *IEEE computational science & engineering 5,* 1, 46–55.

DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., AND ZHANG, C. 2007. Software behavior oriented parallelization. In *PLDI*. 223–234.

DOTSENKO, Y., COARFA, C., AND MELLOR-CRUMMEY, J. 2004. A multi-platform co-array fortran compiler. In *PACT*.

FENG, M., GUPTA, R., AND HU, Y. 2011. Spicec: scalable parallelism via implicit copying and explicit commit. In *PPoPP*. 69–80.

GORDON, M. I., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*. 151–162.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press.

HILFINGER, P., BONACHEA, D., DATTA, K., GAY, D., GRAHAM, S., LIBLIT, B., PIKE, G., SU, J., AND YELICK, K. 2005. Titanium language reference manual. *U.C. Berkeley Tech. Rep., UCB/EECS-2005-15*.

HUDSON, B., MILLER, G. L., AND PHILLIPS, T. 2007. Sparse parallel delaunay mesh refinement. In *SPAA*. 339–347.

INTEL. 2010. Intel STM compiler prototype edition. http://whatif.intel.com/.

KANDEMIR, M., YEMLIHA, T., MURALIDHARA, S., SRIKANTAIAH, S., IRWIN, M. J., AND ZHNAG, Y. 2010. Cache topology aware computation mapping for multicores. In *PLDI*.

KARYPIS, G. AND KUMAR, V. 1998. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput. 48,* 1, 96–129.

KELSEY, K., BAI, T., DING, C., AND ZHANG, C. 2009. Fast track: A software system for speculative program optimization. In *CGO*. 157–168.

KENNEDY, K. AND ALLEN, J., Eds. 2001. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann.

KRISHNAN, V. AND TORRELLAS, J. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput. 48,* 9, 866–880.

KULKARNI, M., BURTSCHER, M., CASCAVAL, C., AND PINGALI, K. 2009. Lonestar: A suite of parallel irregular programs. In *ISPASS*. 65–76.

KULKARNI, M., PINGALI, K., RAMANARAYANAN, G., WALTER, B., BALA, K., AND CHEW, L. P. 2008. Optimistic parallelism benefits from data partitioning. In *ASPLOS*.

LIN, F.-H., WANG, F.-N., AHLFORS, S. P., HÄMÄLÄINEN, M. S., AND BELLIVEAU, J. W. 2007. Parallel MRI reconstruction using variance partitioning regularization. *Magnetic Resonance in Medicine 58,* 4, 735–744.

MELLOR-CRUMMEY, J. M. AND SCOTT, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. acm trans. *Comput. Syst. 9,* 1, 21–65.

MÜLLER, A. AND RÜHL, R. 1995. Extendings high performance fortran for the support of unstructured computations. In *ICS*. 127–136.

QUINLAN, D. 2000. Rose: Compiler support for object-oriented framework. In *CPC*.

REINDERS, J. 2007. *Intel threading building blocks: outfitting C++ for multi-core processor*. O'Reilly Media.

ROGERS, A. AND PINGALI, K. 1989. Process decomposition through locality of reference. In *PLDI*. 69–80.

SCOTT, M., SPEAR, M. F., DALESSANDRO, L., AND MARATHE, V. J. 2007. Delaunay triangulation with transactions and barriers. In *IISWC*.

TIAN, C., FENG, M., AND GUPTA, R. 2008. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*. 330–341.

TIAN, C., FENG, M., AND GUPTA, R. 2010. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI*. 62–73.

UTGOFF, P. E., BERKMAN, N. C., AND CLOUSE, J. A. 1997. Decision tree induction based on efficient tree restructuring. *Mach. Learn. 29,* 1, 5–44.

WEI LIAO, S., DU, Z., WU, G., AND LUEH, G.-Y. 2006. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO*.

ZENG, X., BAGRODIA, R., AND GERLA, M. 1998. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS*.