Shared Work List: Hacking Amorphous Data Parallelism in UPC

Shixiong Xu
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
xushixiong@ict.ac.cn

Li Chen
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
Beijing, China
lchen@ict.ac.cn

ABSTRACT

"Irregular" algorithms using data structures like sparse graphs, trees and sets prevail in the most emerging problems domains such as social network analysis, machine learning, data mining and computational science. The irregularity of underlying data structures leads to unstructured parallelism in these algorithms, consequently making it pretty hard for users to write efficient parallel implementations on distributed memory systems. Unified Parallel C language provides convenience of a global address space with the locality control needed for high performance and scalability. However, the Single Program Multiple Data execution model with a statically fixed set of executing threads makes UPC does not support applications with unstructured parallelism. In this paper, we first put forward Shared Work List to UPC and advocate a programming paradigm for writing applications with amorphous data parallelism on distributed memory systems. We also introduce user-assisted speculative execution based on Active Message model to support speculative execution on distributed memory systems. Efficient mechanism of work dispatching and related optimizations are presented as well. We preliminarily choose Breadth-first Search as a case study to demonstrate the feasibility, programmability and performance benefits out of Shared Work List.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms

Algorithms, Design, Languages, Performance

PMAM 2012 February 26, 2012, New Orleans LA, USA

Keywords

Unified Parallel C, irregular applications, amorphous data parallelism, work list, user-assisted speculative execution

1. INTRODUCTION

"Irregular" algorithms using data structures such as sparse graphs, trees, and sets prevail in the most emerging problem domains such as social network analysis, machine-learning, data-mining, computational science. Due to the irregularity of underlying data structures, parallel irregular algorithms usually reveal unstructured parallelism. Thus, it is exceedingly difficult and error-prone for users to write efficient irregular applications with programming models mainly tailored to "regular" algorithms that use dense arrays, such as finite differences and FFTs. For example, two-sided message-passing model (*e.g.* MPI) makes it hard for users to explicitly manipulate the relationships among the computational units in irregular algorithms.

Partitioned Global Address Space(PGAS) languages improve ease of programming by providing a shared memory abstraction on distributed memory systems. This abstraction offers a convenient programming style, especially for programs with fine-grained data sharing that can be cumbersome in a message passing style. PGAS languages also provide control of data layout and work distribution allowing application developers to take advantage of data locality. Unified Parallel C (UPC), a dialect of C (ISO C99), is one of the most popular languages in the PGAS family. However, UPC uses a Single Program Multiple Data (SPMD) model for parallelism in which a statically fixed set of executing threads run throughout the program execution. It does not directly support applications that involve dynamic unstructured parallelism, such as Delaunay Triangulation Refinement [2].

Keshav Pingali *et al.* [23] put forward a data-centric formulation of algorithms, called the *operator formulation*, in which an algorithm is viewed in terms of its action on data structures. As for graph algorithms, at each executing point, some nodes or edges where computation might be performed are called active elements. The computation *operator* applied to active elements is called activity. This formulation speculatively consider all activities on each active node can execute in parallel similar to conventional data-parallelism. If two activities conflict with each other, some certain mechanisms are introduced to detect conflicts and rollback the performed activities and related data. Under this abstraction of algorithms, amorphous data-parallelism is a general-

ization of the conventional data-parallelism in which (i) concurrent operations may conflict with each other, (ii) activities can be created dynamically, and (iii) activities may modify the underlying data structure. This formulation of algorithms is data centric, similar to the MapReduce model [7], and perfectly matches the PGAS model adopted by UPC.

A natural way to write algorithms with amorphous data-parallelism is to use work lists to keep track of active nodes. Regarding a graph, there are many active nodes and there may be ordering constraints in processing these nodes. In consequence, there are two kinds of work lists, *ordered work list* and *unordered work list*. Respecting the orderings among work items, for a algorithm of which the work list has no work items executing in speculative mode, we argue that all the work items and their orderings during the overall execution actually form an identical graph of the task graph generated by the Cilk-like task mechanism (*e.g.* Cilk [8], PFunc [12], Intel Threading Building Blocks [22], OpenMP 3.0) for the same algorithm. In other words, if there are no speculative work items in a work list, a work list is, in essence, a topological order of tasks in in the task graph out of a given algorithm. Thus, we think, programming in work lists is identical to tasking programming meanwhile in the view of conventional data-parallelism. Therefore, we introduce Shared Work List into UPC to help users to write applications with amorphous data parallelism in the same view of shared memory programming.

Previous work [15] only demonstrated the feasibility of this programming paradigm using work lists on shared memory systems . As amorphous data parallelism requires, the runtime system should support speculative execution to allow operation conflicts to happen between work items. It requires less efforts to implement speculative execution on shared memory systems than that on distributed memory systems due to high communication cost through networks. There is some work on speculative parallelization on distributed systems with software transactional memory (STM) [13]. In our case, we do not intend to automatically parallelize users' code as speculative parallelization does, thereby, some light-weighted speculative execution support is on demand rather than STM. Instead of purely automatic speculative execution as in Gaolis [15], we put forward a semi-automatic user-assisted mechanism for speculative execution to back up the execution of work items. This mechanism asks users to decide which part of data should be considered as the potential conflicting shared data, that is, the shared neighborhoods of several active nodes, and to use necessary language constructs to realize automatic detection of conflicts and rollback of data and computation. With user-assisted speculative execution, by combining the data locality control and partitioned global address space delivered by UPC, we argue that using work lists to write both regular and irregular applications is also a feasible and promising alternative compared with Cilk-like task-based methods.

In this work, we make following contributions:

1. We put forward **S**hared **W**ork **L**ist (SWL) to UPC to tackle amorphous data-parallelism and implemented it in the UPC translator and runtime system.
2. We present a runtime system for SWL by introducing a user-assisted speculative execution model and a communication optimized mechanism of dispatching work items.
3. To our best knowledge, it is the first try to take advantage of Active Message to realize user-assisted speculative execution on distributed memory systems.
4. We demonstrate that using our proposed programming paradigm, for Breadth-first Search (BFS), users can easily achieve comparable performance and scalability as MPI at a low cost of programming efforts.

The rest of this paper is organized as follows. Section 2 reviews some related work on asynchronous execution in UPC and speculative execution on distributed memory systems. We introduce Shared Work List (SWL) in Section 3 and present the implementation and optimization of it in Section 4. We preliminarily choose Breadth-first Search as a case study to demonstrate our proposed programming paradigm in Section 5. The experimental results and conclusion are given in Section 6 and Section 7, respectively.

2. RELATED WORK

2.1 Asynchronous Execution in UPC

S. Olivier *et al.* [20] give Unbalanced Tree Search (UTS) benchmark and demonstrate the performance of asynchronous work-stealing of dynamic load imbalance in UPC. In [24], *Asynchronous Remote Methods* is put forward and its performance is demonstrated by a nested, tree based code — MADNESS. Jithin Jose *et al.* propose *UPC queues* for graph applications (*i.e.* BFS, UTS) [11]. Our Shared Work List and its related operations are similar to the above two methods. In contrast, our approach is a unified programming paradigm and its runtime supports speculative execution besides asynchronous execution.

S.J Min *et al.* [18] put forward a task library named Hot-SLAW, which extends the scalable locality-aware adaptive work-stealing scheduler (SLAW) [9] on shared memory systems. Their work introduces a *hardware topology-aware hierarchical victim selection strategy* and a *hierarchical chunk selection approach* to optimize the performance of work-stealing on distributed memory systems. They choose four irregular applications, Fibonacci, N-Queens, Unbalanced Tree Search (UTS) [19] and SparseLU to demonstrate the performance of their proposed work scheduler on both shared memory systems and distributed memory systems. Their work shows competitive performance against OpenMP on shared memory systems. But for distributed memory systems, as they only give the performance of their proposed task scheduling trade-offs and most of the applications they choose are not data-intensive, they do not prove the overall performance benefits of Cilk-like tasking. Though they put forward a hierarchical work stealing, the interfaces for dispatching tasks do not include any data-thread affinity hints, which plays a critical role in the data-intensive applications, such as BFS. Our proposed Shared Work List, in essence, is equal to this tasking mechanism. Meanwhile, instead of the passive mode in the Cilk-like tasking, we adopt an active mode, that is, when a thread has no work to do, it waits until other threads actively pass a task to it. As the data-thread affinity information is given at the point of dispatching a work item, a task can execute at the most appropriate place taking data locality into account.

2.2 Irregular Applications in UPC

In [6] [5], a fast PGAS implementation of distributed graph algorithms is presented, including connected components and minimum spanning tree problems. Junchao Zhang *et al.*

[25] give the Barnes Hut algorithm [3] in UPC and apply successive optimization techniques to considerably improve performance. The above work mainly focuses on data locality optimization. Our SWL can automate part of their manual optimization. C.M. Maynard [17] presents the performance comparison of distributed implementations of hash table in UPC and one-sided MPI. His work shows that the UPC implementation of the distributed hash table runs faster than MPI and scales better with the number of processing elements for both weak and strong scaling.

2.3 Speculative Execution on Distributed Memory Systems

Distributed software transactional memory systems (DSTM) aim to support transaction execution on systems without shared memory and in turn give support for speculative execution. Distributed Multi-versioning (DMV) [16] modifies a software distributed shared memory system (SDSM) to support transactions. Cluster-STM [4] is an software transactional memory system for large-scale clusters. DiSTM [14] is a DSTM system which builds on Java Remote Method Invocation (RMI). DiSTM detects and resolves conflicts at object granularity. Distributed Software Multithreaded Transactional memory system (DSMTX) [13] extends the above systems with Multi-threading Transactions (MTXs) to support Spec-DSWP [10]. All these systems expose a unified virtual address space to programs. As for our Shared Work List, we do not intend to build a system for speculative parallelization from a sequential program. The speculative execution is just a key component for exploiting more parallelism in a already parallelized application. Hence, we choose the light-weighted user-assisted speculative execution which not only conforms to the original semantics of memory operations in UPC but also gives users rights to control the speculative behaviors of their applications.

3. SHARED WORK LIST

In this section, we first discuss the design requirements for Shared Work List (SWL) in UPC and explain how these can be satisfied in our proposed SWL.

3.1 Design Requirements

We expect that a design for Shared Work List in UPC should satisfy the following requirements.

Programmability. Ease of programming is an important reason for the growing popularity of PGAS languages in general and UPC in particular. Ensuring this is imperative for the acceptance of any new extensions to the UPC specification. Thus, we consider this first requirement while proposing SWL in UPC. Shared Work List can improve programmability for irregular applications with amorphous data parallelism by concise and intuitive algorithm expression in global-view similar to programming on shared memory systems.

Flexibility. On distributed memory systems, communication often takes a large proportion of the overhead of parallelization. To exploit data locality, users should have rights to control the behavior of adding a work item according to the context of this operation. For example, if all the data needed by the construction of a new work item is local to the executing processor, the work item is best to be locally built; if most of the data required is on the remote side, it is reasonable to migrate the work constructor to the remote side and build the work remotely but locally to the remote side. Moreover, there are some cases where users want to customize the way in which a work item comes into a work list as the algorithm requires. Our proposed Shared Work List gives users interfaces to control the behavior of adding a work item such as data-thread affinity, user-defined work constructor, and it also supports algorithm-level optimization such as manual elimination of redundant work items.

```
typedef int Work_t;
typedef struct parameter Msg_t;

/*declare work lists with the type of a work item*/
shared worklist list1(Work_t);
shared worklist list2(Work_t);
shared [BLKSIZE] int data[BLKSIZE*THREADS];
/*user-defined work constructor to upc_worklist_add*/
Work_t usr_add(Msg_t msg){
    handling msg
    /*customize the behavior of adding a work item*/
    Work_t res_work =  ...
    return  res_work;
}
/*non-blocking operation*/
upc_worklist_foreach(Work_t w: list1)
{ /*enter work region*/
    Msg_t msg;
    processing work w
/*upc_worklist_add(worklist, affinity_expr,
     work_constructor)*/
    upc_worklist_add(list1, &data[i], usr_add(msg));
}/*finish work region*/
/*blocking operation*/
upc_worklist_until(Work_t w:list)
{ /*enter work region*/
    Msg_t msg;
    processing work w
    upc_worklist_add(list, &data[i], usr_add(msg));
}/*leave work region*/
```

Figure 1: Code examples using Shared Work List

Speculative Execution. Speculative execution is a critical part for running applications with amorphous data parallelism. However, it may bring significant overhead due to communication cost on distributed memory systems. Thus, a light-weighted mechanism for speculative execution on distributed memory systems is on demand. Differing from the traditional speculative parallelization from a sequential program to a parallel one, speculative execution in the scenario of work lists merely helps users to exploit more parallelism in a parallel program. Not all irregular applications can reveal more parallelism by running in the speculative way. Our Shared Work List currently only introduces user-assisted speculative execution, where users is in charge of specifying the speculative behaviors of a work item and the underlying runtime gives execution support for these speculative behaviors.

3.2 Language Constructs

3.2.1 Declaration of Shared Work Lists

We propose a language extension that can declare a SWL with work items in the data type *work_item_t*, as follows:

```
shared worklist list_name(work_item_t);
```

, where *worklist* is a newly added key word to UPC. The SWL declared is by default with a capacity of infinite work

items. Users don not have to concern about the sizes of SWL, as it can automatically adjusts the size with regard to the dynamic usage at runtime. A declared SWL is, in fact, treated as a shared distributed list. However, to hide the complexity of manipulating the list, users can only orchestrate the SWL with our predefined operations in the following section.

3.2.2 Manipulation of Work Items

There are two language constructs for users to add a work item into a specified SWL, as follows:

```
upc_worklist_add (SWL identifier,
  affinity expression, work constructor);
upc_worklist_add (SWL identifier,
  affinity expression, work item);
```

Besides *SWL identifier* which specifies the target SWL, both constructs require users to specify the data-thread affinity to tell the runtime system where this work item will reside. On distributed memory systems, it is critical to perform local computation as fast as possible with minimum communication. Thus, the data-thread affinity information helps the runtime system to distinguish the local and remote work items and to dispatch each work item to the most appropriate UPC thread. Users have two options to add a work item. If a work item is constructed locally, which may requires little communication, the second *upc_worklist_add* is recommended, as it just transfers the work item to the target SWL. If a work item is best to be constructed on the remote side, perhaps, the construction of this work item requires remote data, or, users want to place some actions before it comes into the SWL, users may choose the first *upc_worklist_add* by giving a function as a work constructor with the type of work item as the return type. These two options give users great flexibility to control the behavior of adding a work item into a SWL. To support the amorphous data parallelism, we supply the following two parallel work item iterators to help users access each work item in the functional programming style (*i.e.* MAP).

```
upc_worklist_foreach(work_item_t w: SWL){ ... }
upc_worklist_until(work_item_t w: SWL){ ... }
```

As our proposed SWL is an unordered work list, the two iterators can access the contained work items in any order. Note that, these two iterators work in the collective way. That is, all processors must collectively start these two iterators after a global synchronization and leave these two iterators with a global synchronization; if and only if when there are no work items are available in the SWL, a processor can then leave the iterators. In order to support ordered work lists, we design that the two work item iterators work in different modes. *upc_worklist_foreach* works in a blocking mode, that is, no new work items are allowed to be appended to the same SWL as the iterator works. In contrast, *upc_worklist_until* allows new work items to be added to the same working SWL in flight. The former one is useful when it comes to the case of achieving ordered work lists. In this case, users can build several SWLs, dispatch work items into different SWLs, and add necessary synchronizations in order to preserve the desire executing orderings among the work items, as demonstrated by the level-synchronized Breadth-first Search (BFS). The *upc_worklist_until* iterator greatly helps asynchronous algorithms, such as the asynchronous BFS based on the Single Source Shortest Path.

There are constraints on synchronizations within the scope of the executing body of each work item. Collectives operations (*e.g. upc_barrier*, memory allocations on the shared data space), fine-grained synchronization operations like locks are prohibited. I/O operations are not allowed at present to simplify speculative execution. Besides, asynchronous memory operations are required to be finished before leaving the scope. Even though these two iterators look like the traditional *for* and *while* loops, only *continue* is allowed to terminate the current work item while other operations like *break* and *goto* are forbidden.

In order to help users to achieve ordered work lists by using unordered work lists, the following collective operation is introduced to exchange two given SWLs. This operation is used in the level-synchronized BFS which will be presented in Section 5.3.2.

```
upc_worklist_exchange(SWL identifier1,
       SWL identifier2);
```

3.2.3 User-assisted Speculative Execution

Speculative execution is an essential part for amorphous data parallelism, which allows conflicts between operations to happen within different work items. We put forward some language constructs to support semi-automatic user-assisted speculative execution, in which the users are in charge of specifying the speculative behaviors of a work item and the runtime system takes the responsibility of detection of conflicts and rollback of data and computation. We will discuss the role of cooperations between users and runtime system more in detail in Section 4.2.

In UPC, shared data between processors can be accessed either by a reference of a shared array or by a shared pointer of a specific data type. Within a certain processor, though work items belonging to this processor share the data local to this processor, the sequential executing ordering of these work items ensures that no operation conflicts will occur. Thus, we assume that only operations on shared data structure can conflict with others from different work items. When a work item speculatively operates on a shared data item, it first has to obtain the ownership of the desired data before conducting any computations on it. Then, after the computation, the work item release the ownership back to the original owner. If two work items are trying to get the ownership of the same shared data, a conflict happens and the runtime system detects this conflict and rollbacks the computation and data of a chosen work item according to a predefined rule. Note that, not all the shared data should be necessarily accessed in speculative state. We could aggressively treat all the references of shared data in speculative state, but doing this would lead to a large amount of unnecessary overhead of manipulating speculative data due to the high communication cost on distributed memory systems. Thus, to distinguish the shared data accesses in speculative state from the unprotected ones, we supply the following constructs, *speculative assistant operations*, to achieve user-assisted speculative execution:

```
upc_spec_get(void * restrict dst,
  shared const void * restrict src, size_t n);
upc_cmt_put(shared void * restrict dst,
  const void * restrict src, size_t n);
```

upc_spec_get is used to obtain the ownership of desire shared data. Besides the semantics of the *upc_memget()*, it imposes a fine-grained lock upon the shared data. When two work items are trying to lock the same shared data simultaneously, a conflict happens. To enable future rollback of data in case of conflicts, this construct makes a shadow local copy of the original shared data. No matter how this copy of data will be accessed, read, write, or both, once conflicts are detected, the runtime system can just rollback this work item by releasing the ownerships of shared data it has obtained and discarding all the local copies directly. It is users' responsibility to specify which data is assumed to be operated in speculation prior to the execution.

upc_cmt_put gives users control to commit the data in the speculative state. Only if all the *upc_spec_get* operations before the this operation finish successfully, we can say that this work item wins in the race of speculative computation and is able to conduct the remainder computation. When all the computation completes, a winer work would commit the shadow copy to the corresponding shared data and return the ownerships of the shared data it has acquired. Other work items which fail in the race of speculative computation may then restart the work as the winer does.

4. IMPLEMENTATION

In this section, we first give the description of the execution model for Shared Work List. More detailed discussions and implementing strategies on user-assisted speculative execution and Active Message-based mechanism of work dispatching are also presented. We implement our proposed Shared Work List in the Berkeley UPC translator and runtime 2.14.

4.1 Execution Model of Shared Work List

Differing from the master-slave execution model for work lists in Galois, the execution of Shared Work List (SWL) employs the Single Program Multiple Data (SPMD) style like *upc_forall*. The execution model of the SWL in the case of *upc_worklist_until* operation is shown in Figure 2. Before the start of a SWL iterator, all processors, that is, UPC Threads, have to collectively enter the work region indicated by the body of the iterator. Once all threads have been in the work region, each thread initiates its state as *executing*. All the work items belonging the local part of a SWL in a processor execute sequentially whereas different work items belonging to different processors proceed in parallel. The user-assisted speculative execution is designed for work items working in parallel, more details are in Section 4.2. When a work item is being deposed from a SWL, new work items can be spawned within that work to a target SWL either synchronously or asynchronous. Coalescing is an option for users to optimize the performance of communication. When a thread empties its local part of the given SWL, it changes its state into *idle* and enters work termination detection by detecting whether there are still any work items in the global SWL. If there are no available work items, the thread leaves the work region; otherwise, it waits for new work items to come into the local part of the SWL, or the empty of the global SWL.

4.2 User-assisted Speculative Execution

Figure 3 gives an example of user-assisted speculative execution. In this example, the shared data are distributed over three threads and all these threads are dealing with

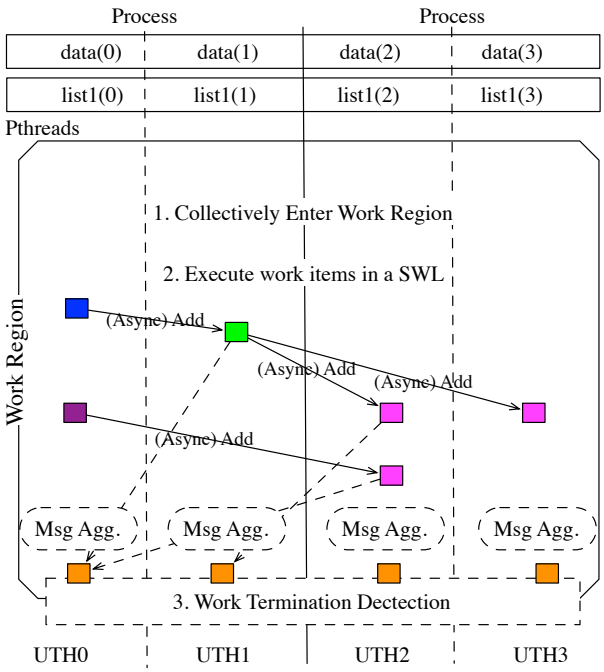

Figure 2: The work region of a *upc_worklist_until* iterator

work items in the given SWL in the speculative mode. Here, we assume that operations (in thread 0) on shared data *d2* conflict with the operations (in thread 2) on *d3*.

4.2.1 Translator

Identification of Speculative Behavior. As speculative execution requires support from runtime system for detection of conflicts and rollback of data and computation, it is unwise to assume our proposed SWL iterators always work in the speculative way. Thus, the translator first checks whether there are any *speculative assistant operations* (*upc_spec_get* or *upc_cmt_put*) within the scope of a given SWL iterator. If yes, the translator gives a hint to the runtime such that this SWL iterator will work in speculative mode. Otherwise, this SWL iterator works like traditional parallel forall loops. Since there may be function calls within the scope of a given SWL iterator, inter-procedural analysis is applied to detect the existence of speculative assistant operations for the work item iterator to decide work either in speculative mode or non-speculative mode.

Fine-grained Atomic Protection. To enable acquisition of ownerships of shared data to be operated in speculative mode, it is intuitive to use the high-level UPC shared lock and its related operations. However, it is not wise to use this shared lock to achieve atomic protection in the case of speculative execution. On one side, for fine-grained data accesses, it is inefficient to allocate as many shared locks as the number of the data items as this would waste the space of the shared data among processors. What is more, unlike traditional lock and its operations on shared memory systems (*e.g.* Pthreads), operations on a shared lock are actually message communications between processors which will inevitably increase the overhead of speculative execu-

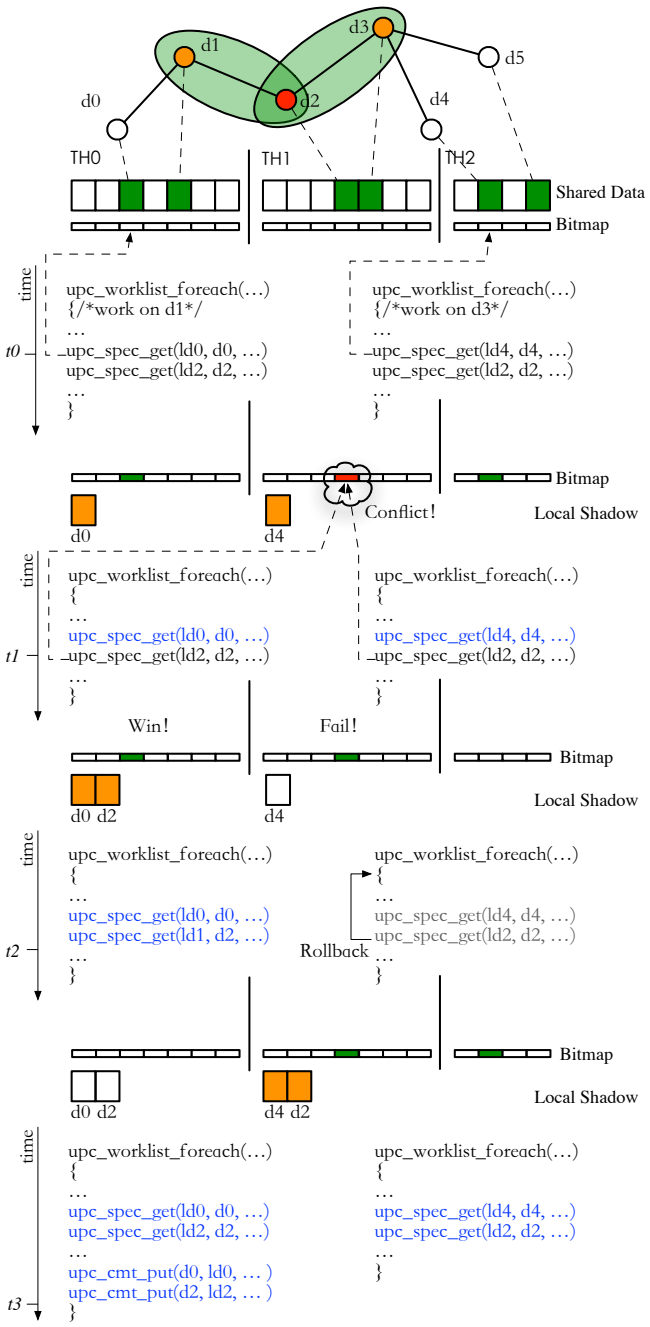

Figure 3: An example of user-assisted speculative execution

tion. Instead of the shared lock, we employ a fine-grained synchronization mechanism of Full/Empty Bits (FEB) to achieve atomic protection. The translator first collects all the data references of *upc_spec_get()* operations and allocates a bitmap array for each shared array accordingly. The operations on these bitmap-based locks are achieved through Active Message. When a work item executes a *upc_spec_get()* operation, for example, the runtime system first sends a AM message to the target processor as indicated by the data-thread affinity of the shared array references. When the target processor receives this message, it checks the status of the bit in the bitmap array for this shared array reference. If this bit is empty, that is 0, the target processor sets it to 1, and replies a success message to the sender processor. Then, the following construction of local shadow copies of the shared data can proceed. Otherwise, the target processor directly reply a failure message to the sender processor. When this message arrives at the sender, the runtime treats this message as a hint of conflicts. As a result, the runtime system rollbacks the "dirty" data and speculatively completed operations. The operations on bitmap arrays are local to a certain processor, thereby, the bitmap array can be just locally allocated without taking any space in the shared data space.

Correctness Checking. When a SWL iterator works in the speculative mode, each work item executes as if it was protected by a critical section. The first *upc_spec_get()* occurring in the scope of a work item indicates the start of a critical section, and the last *upc_cmt_put()* operation ends the critical section. To avoid deadlock, all *upc_spec_get()* operations must come before *upc_cmt_put()* operations. That is, these two kinds of operations cannot interleave with each other. Furthermore, each *upc_spec_get()* operation should be matched by a *upc_cmt_put()* operation with the same region of shared data as the *lock()* and *unlock()* operations do. The translator statically checks this two constraints for users. If there are any violations to these rules, the translator treat these as compilation errors. Note that, predicated data flow analysis and inter-procedural analysis is a necessity for checking the validity for speculative execution.

Optimizing Local Shadow Copies. There are possibly some cases in which some work items are only trying to modify the shared data without reading the original data or to load the original data without changing it in the end. But according to our proposed user-assisted speculative execution, users still have to use the *upc_spec_get()* operation to create a local shadow copy of the shared data by transferring the desired data from target processor in order to acquire the ownership of the data. In fact, the runtime can just allocate a local temporary data item as the local shadow copy for the local read without writing back or further write operations without loading the data. Unnecessary data transfers would incur the waste of network bandwidth and increase the overhead of speculative execution. The translator conservatively optimizes these operations with the help of the def-use chain obtained from both intra-and inter- procedural data-flow analysis. If there are no reaching uses from a created local shadow copy, then the unnecessary data transfers can be eliminated and the *upc_spec_get()* operations just work for the acquisition of ownerships of the desired data. Since a local shadow copy may be accessed through pointers, data-flow analysis may involve pointer-analysis, which may not be precise enough and lead to the limited applicability of this optimization .

4.2.2 *Runtime*

Rollback of Data and Computation. Since users have given hints to the runtime system about which shared data should be protected and operated in the speculative state, the runtime system maintains an operation table and logs all the speculative assistant operations in the user defined order for each processor. Note that, the manipulation of tables for speculative assistant operations runs in the Single Pro-

gram Multiple Data way instead of master-slave mode which may constrain the scalability on distributed memory systems. When a *upc_spec_get()* operation fails, which means a conflict happens, the runtime system is triggered to rollback the data and computation by just doing corresponding *upc_cmt_put()* operations but without any data transfers for the already created local shadow copies resulted from all the operations already logged in the table. In other words, the runtime system just discards the local shadow copies of the protected shared data and releases the ownerships of these data items.

4.3 Distributed Work Dispatching

As a Shared Work List is actually a distributed list over processors, adding a work item into a portion of the list belonging to a specific processor requires communication through network. In addition, two work items may concurrently compete with each other to get into the same list, therefore, a synchronization mechanism is needed to achieve the mutual exclusion of these operations of adding work items. UPC standard only supplies shared locks for users to achieve mutual exclusion. However, locks on shared-memory systems are infamous for contention and come with considerable overhead on distributed memory systems, thereby, they are not scalable. We observe that two messages arriving at the same processor cannot be real concurrent due to the serialization of messages out of the network port. UPC runtime relies on Active Message supplied by the GASNet interface to realize communications and remote memory operations. We utilize Active Message to implement distributed work dispatching and related optimizations.

AM-based Distributed Work Dispatching. Active Message is basically a mechanism of remote procedure call, where each message contains at its head the address of a user-level handler and the arguments in its message body, and the the function will be executed on message arrival. Using this mechanism, more control can be given to the user on how to build and push a work item into a SWL. As stated in Section 3.2.2, there are two candidate operations for adding a work item into a SWL. One of them allows users to give a user-defined work constructor function. When a function call is given in the work adding operation, the translator first checks whether this function has the same return type as a work item defined in the declaration of the enclosed SWL. If this function is qualified as a work constructor, the translator registers it in the customized handler id table. When an AM comes with this handler id, the receiver will execute the right handler function according to the handler id, and produce a work item. To build a work item at the remote side or transfer an available work item, we use the GASNet Active Message interface *gasnet_AMRequestMedium(...)*, which carries payload along with the handler id and arguments.

Coalescing Work Items. Coalescing avoids the communication cost for each work adding operation. Multiple work items destined for the same remote thread are aggregated and are sent as a single active message. In order to support coalescing, separate work item buffers are kept for each of the remote UPC threads. As only one work item buffer is necessary for each remote thread, the memory consumption of these buffers is not considerable. These buffers are created when all threads collectives enter the work region of a given SWL iterator with a coalescing size specified. User can change the coalescing size only through an environmental variable *GASNET_WORKLIST_COALESCE*.

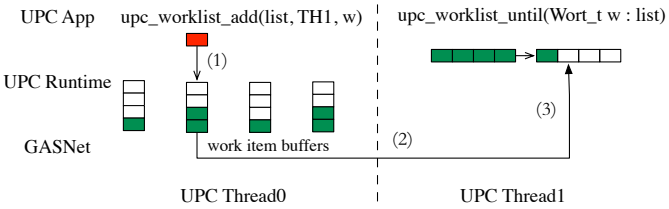

Figure 4: Active Message-based work adding operations

As shown in Figure 4, within a work region of a SWL iterator, a work item is put into the work item buffer designated for the destination thread. The work item is sent out, when the buffer is full, or when a thread is trying to leave the work region. If the work adding operations are invoked without coalescing, the work item is sent out immediately.

Asynchronous Work Dispatching. When a work item is attempting to spawn a new work item into a given SWL, the corresponding working thread has to be waiting until this work item is actually in the target SWL, which may significantly bring down the throughout of disposing work items in the SWL. Like asynchronous *upc_memput()* operations, when a work adding operation has been encapsulated into a message and successfully injected into the network, we can deem this operation has finished. To ensure that the work item asynchronously dispatched is eventually added into the target SWL, the runtime maintains a table for these work adding operations. When a work adding operation is triggered, the runtime keeps a record for it. Afterwards, the runtime checks the status of these operations and waits until all the operations checked are finished when the logging table is full or when all threads are trying to leave the work region. Asynchronous work dispatching helps overlap communication with computation, and in turn further reduces the overhead of work dispatching on distributed memory system on top of work item coalescing.

5. A CASE STUDY: BREADTH-FIRST SEARCH (BFS)

To demonstrate the feasibility and performance of our proposed Shared Work List (SWL), we initially choose Breadth-first Search (BFS) as a case study, of which different implementations require different types of work item iterators on SWLs.

5.1 Selection of BFS algorithms

Level-synchronized Style. Breadth-first Search of undirected graph starts from a root, and explores all the neighbors of this node before exploring their neighbors, etc. Hence, it is natural to implement BFS in a *level-synchronized* way. Within a single computation level, the work items produced from the previous level are dispatched among all UPC threads and new items are generated for the next level. When there is no more new work items available, the whole BFS terminates. Synchronizations are required between two successive computation levels to ensure the correctness of this algorithm.

Asynchronous Style. Roger Pearce *et al.* [21] put forward an asynchronous BFS based on the asynchronous Sin-

gle Source Shortest Path (SSSP) on shared memory systems. The main idea of their asynchronous approach is to allow multiple SSSP algorithms to execute in parallel with the help of prioritized visitor queues. Like Bellman-Ford, their approach relies on *label-correcting* to compute the traversal, and completes when all corrections are complete. Like Dijkstra's SSSP, their approach traverses paths in a prioritized manner, visiting the shortest path possible at each visit. Their approach does not introduce synchronizations between steps at the costs of multiple visits per vertex. BFS can be computed by applying their asynchronous SSSP algorithm with all edge weight equal to 1.

5.2 Graph 500

Graph 500 Benchmark Specification [1] is proposed to direct design of a new set of benchmarks that can evaluating the performance of supercomputers in the context of data-intensive applications. Graph 500 benchmark consists of three comprehensive benchmarks to address application kernels: Search (Concurrent Search), Optimization (Single Source Shortest Path), and Edge Oriented (Maximal Independent Set). Concurrent search benchmark consists of three phases (termed as kernels in benchmark specification). The first is "Graph Construction", which generates edges. The "Kronecker Generator" algorithm is used in the reference implementation. From the edge list, a graph is contracted in Compressed Sparse Row (CSR) format or Compressed Sparse Column (CSC) format. The second kernel is the actual "Breadth-first Search" with 64 search keys randomly sampled from the vertices in the graph. For each search key, BFS traversals are made one by one. The validation kernel, the final kernel, ensures the correctness of the 64 BFS traversals.

The Graph 500 benchmark package from its official website contains several versions of referenced implementation in MPI, including MPI_one_sided, MPI_simple, and MPI_replicated versions. All these versions are in the level-synchronized style but with different implementation strategies for queue operations. The MPI_one_sided uses a three-color scheme to present queue and utilizes one-sided communication delivered in MPI 2 to achieve queue operations at the cost of redundant computation. The MPI_simple implements a distributed queue-based BFS with message aggregation. This version is the most similar to the sequential BFS. In the MPI_replicated, every thread has a copy of global queue. Moreover, within a phase, each thread makes contributions to the global queue through collective operations, which prohibits overlapping communication with computation.

5.3 Implementations in UPC

In this work, we focus on the second kernel of Graph 500. As it does not give any referenced code in UPC, we first rewrite the Graph 500 in UPC. For simplicity, we keep the the generation of edge lists, transformation from edge lists to CSR data format, and validation in MPI. Each UPC thread locally keeps the adjacent list information of vertices that it owns as the MPI does. The key data structures for the algorithm shared by all the processors (*e.g.* Pred) are declared as the UPC shared arrays . As stated in Section 5.1, we can have different versions of BFS in UPC.

5.3.1 Global-view BFS

We give the first two implementations of BFS referring to the MPI_one_sided version in the referenced Graph 500 code. These two versions are written in the global-view with *upc_forall*. Compared with MPI_one_sided version, when writing global-view applications in UPC, users do not have to orchestrate the communication buffers, which are required by the one-sided operations on *MPI_Window* in MPI. Note that, UPC language only supports for fine-grained accesses to shared data. There are no operations like *MPI_Accmulate* in UPC, which not only access to the shared data, but also carry necessary arguments to conduct a specific computation (*e.g.* ADD, MAX, MIN) at the same time. Achieving the same operation in UPC results in extra communication overhead introduced by the redundant data transfers, as shown in Figure 5. We consider adding this kind of accumulating operations to UPC as our future work to better UPC for global-view applications.

```
MPI_Win_create(pred2, ...., &pred2_win); /*comm. buffers*/
MPI_Accumulate(&local_vertices[v_local], ...., MPI_MIN, pred2_win);
```

(a) MPI One-sided Operations

```
shared [...] long  pred2[...]; /*implicit comm. buffers*/
pred2[pos] = long_min(pred2[pos] , local_vertices[i]); /*redundant comm.*/
```

(b) UPC

Figure 5: Comparison of one-sided operations

Both UPC_Global and UPC_Global_Bitmap adopt the level-synchronized algorithm and a three-color scheme to implement queue operations. For each computation level, redundant computations are introduced to get the current input nodes produced from last level. Bitmap is added with the intent of reducing these redundant computation. However, as the experimental results in the Section 6.2 show, the introduction of bitmap degrades the performance of BFS as it nearly doubles the communication.

5.3.2 Level-synchronized BFS in SWL

We also give a level-synchronized BFS , SYNC_SWL, in the SWL referring to the MPI_Simple version. As our proposed SWL only designed for unordered work lists, two SWLs are needed to preserve the orderings between different computation levels.

Writing level-synchronized BFS in SWL helps to achieve optimal work construction by migrating the construction of a work item for the next computation level to the most appropriate UPC thread taking into account the communication overhead. Moreover, user-defined work constructors realize the monotonicity of updates to the nodes due to the serialization of messages through network. The monotonicity of the updates also achieves the atomicity of the updates to the same node and eliminates the necessity of high-cost user-assisted speculative execution. In addition, as stated in Section 4.3, the runtime system can achieve automatic message aggression for applications in SWL. Compared with the explicit and complex optimization done in the MPI_Simple version by hand, writing applications in SWL can helps users to achieve nearly the same optimization while at a low cost of programmability.

5.3.3 Asynchronous BFS in SWL

The asynchronous BFS in SWL adopts the aforementioned asynchronous BFS based on the asynchronous SSSP. Differing from the SYNC_SWL, only one SWL is enough for

the overall computation. Similar to the SYNC_SWL, we use user-defined work constructor to build work items locally. Besides, in order to cut off the infeasible SSSP paths as early as possible and to prevent the explosion of the work list, we introduces two optimizations in the work constructor: 1) instead of directly creating a new work item and pushing it into the work list, we first check whether it is necessary to create a work item according to the current path from the source vertex carried by the constructor's argument; 2) we introduce a bitmap array to record whether there is already a work item for a vertex in the work list. If the answer is positive, when creating a new work item, we compare the two work items, and replace the existing one with the best one in the work list. Otherwise, the new created work item is directly pushed into the work list. The ASYNC_SWL also benefits from the automatic message aggression and the elimination of user-assisted speculative execution thanks to the monotonicity of the updates to the vertices.

6. EXPERIMENTAL RESULTS

6.1 Experimental Platform

We used an Intel cluster for our experiments. This cluster consists of 16 computing nodes with 8 Intel Xeon X7550 8-core processors, operating at 2.0 GHz,. Each node has 256GB of memory and is equipped with MT26428 QDR Connect X HCAs (40Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 36-port Mellanox QDR switch. The operating system used is CentOS release 5.3, with kernel version Linux 2.6.18-128.el5 and OpenFabrics version 1.5.

6.2 Graph 500 Benchmark Performance

We tested the benchmark for an input graph with 1 million vertices and 16 million edges, for varying number of systems sizes, 64, 128, 256, 512 UPC-threads. We conducted this experiment with the InfiniBand GASNet conduit for our altered UPC runtime with support for SWL.

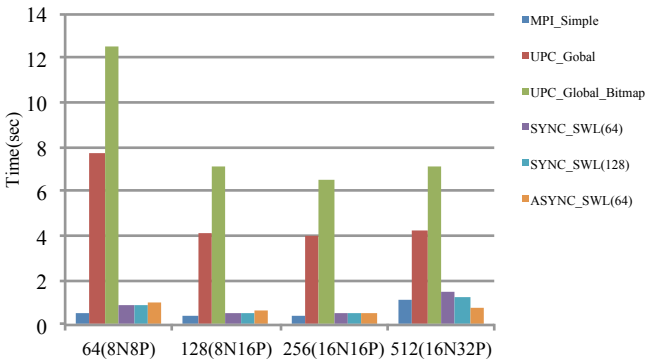

Figure 6: Performance of BFS (Strong Scaling, 1 Million Vertices)

Performance results of Graph 500 benchmark are presented in Figure 6. To demonstrate the performance benefits of our proposed SWL, we only compared the performance of our implemented BFSs with the MPI_Simple. As stated in Section 5.3.1, UPC_Global_Bitmap considerably degrades the performance of the UPC_Global due to the extra communication for the operations on bitmap arrays. The SYNC_SWL versions nearly have the same performance and scalability as the MPI_Simple. Different sizes for coalescing work items hardly make any differences in the the performance of the SYNC_SWL on a small scale of UPC threads. When it comes to 512 UPC threads, the SYNC_SWL with coalescing size 128 performed better than the one with coalescing size 64. These results demonstrate that writing level-synchronized BFS in SWL can help users to achieve nearly the same performance and scalability as MPI while at a low cost of programmability. As for the ASYNC_SWL, the performance of it highly relies on the structure of the underlying graph. The ASYNC_SWL performed a little worse than the SYNC_SWL version on a small scale of UPC threads. This is possibly because that the redundant computation introduced by multiple speculative SSSP algorithms takes more time than the time of synchronization in the SYNC_SWL. Surprisingly, the ASYNC_SWL greatly outperformed both the MPI_Simple and the SYNC_SWL on the 512 UPC threads. The reason for this is perhaps that the critical path of computation steps in the asynchronous BFS is smaller than that of the level-synchronized BFS, and the synchronization overhead increases as the computing processors scale.

7. CONCLUSION

This paper proposes Shared Work List to tackle amorphous data-parallelism in UPC. It naturally augments the work distribution constructs (*i.e.* upc_forall) in UPC and gives users a global-view programming paradigm like programming on shared memory systems. The two work item iterators proposed can help users to write different types of applications requiring either unordered work list or ordered work list. The Active Message model based semi-automatic user-assisted speculative execution allows the occurrence of operation conflicts and accordingly it plays a critical role in supporting applications with amorphous data parallelism. To our best knowledge, it is first time to introduce Active Message-based speculative execution in a PGAS language. Moreover, the Active-Message based work dispatching mechanism not only gives users great flexibility to control the behavior and location of adding a work item, but also achieves the automation of manual optimization on communication, such as asynchronous data transfers and message aggregation.

As the preliminary experimental results from a case study of Breadth-first Search show, writing applications with Shared Work List can help users to achieve satisfactory performance while at the ease of programming. In the future, we will explore more applications with amorphous data parallelism and demonstrate the performance benefits of our proposed solution.

8. ACKNOWLEDGMENTS

This research was supported in part by the National Hi-Tech Research and Development 863 Program of China under Grant No. 2009AA01A129, the National Fundamental Research Program of China (2011CB302504) and the Innovation Research Group of NSFC (60921002).

9. REFERENCES

[1] Graph500 specification, 2011.
[2] C. D. Antonopoulos, X. Ding, A. Chernikov, F. Blagojevic, D. S. Nikolopoulos, and

N. Chrisochoides. Multigrain parallel Delaunay Mesh generation: challenges and opportunities for multithreaded architectures. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. ACM, June 2005.

[3] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6270):446–449, 1986.

[4] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 247–258, New York, NY, USA, 2008. ACM.

[5] G. Cong and G. Almasi. Fast PGAS connected components algorithms. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 13:1–13:6, New York, NY, USA, 2009. ACM.

[6] G. Cong and G. Almasi. Fast PGAS implementation of distributed graph algorithms. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[7] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[9] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, New York, NY, USA, 2010. ACM.

[10] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[11] J. Jose, S. Potluri, M. Luo, S. Sur, and D. K. D. Panda. Upc queues for scalable graph traversals: Design and evaluation on infiniband clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Oct. 2011.

[12] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. Pfunc: modern task parallelism for modern high performance computing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 43:1–43:11, New York, NY, USA, 2009. ACM.

[13] H. Kim, A. Raman, F. Liu, J. Lee, and D. August. Scalable Speculative Parallelization on Commodity Clusters. *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 3–14, 2010.

[14] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. DiSTM: A Software Transactional Memory Framework for Clusters. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 51–58, 2008.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *Communications of the ACM*, 52(9), Sept. 2009.

[16] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 198–208, New York, NY, USA, 2006. ACM.

[17] C. Maynard. Comparing upc and one-sided mpi: A distributed hash table for gap. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Oct. 2011.

[18] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, Oct. 2011.

[19] S. Olivier, J. Huan, J. Liu, J. Prins, and J. Dinan. UTS: An unbalanced tree search benchmark. *Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2007.

[20] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 123–131, 2008.

[21] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[22] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23:298–298, April 2008.

[23] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25, New York, NY, USA, 2011. ACM.

[24] A. Shet and V. Tipparaju. Asynchronous programming in UPC: A case study and potential for improvement. *In the 1st Workshop on Asynchrony in the PGAS Programming Model (APGAS), collocated with the 23rd International Conference on Supercomputing (ICS)*, 2009.

[25] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM Request Permissions, Nov. 2011.