Asynchronous Programming in UPC: A Case Study and Potential for Improvement *

Aniruddha G. Shet, Vinod Tipparaju, and Robert J. Harrison

Oak Ridge National Laboratory
PO Box 2008, Oak Ridge, TN 37831 USA
{shetag,tipparajuv,harrisonrj}@ornl.gov

Abstract. In a traditional Partitioned Global Address Space language like UPC, an application programmer works with the model of a static set of threads performing locality-aware accesses on a global address space. On the other hand, asynchronous programming provides a simple interface for expressing the concurrency in dynamic, irregular algorithms, with the prospect of efficient portable execution from sophisticated runtime schemes handling the exposed concurrency. In this paper, we adopt the asynchronous style of programming to parallelize a nested, tree-based code in UPC. To maximize performance without losing the ease of application programming, we design Asynchronous Remote Methods as a potential extension to the UPC standard. Our prototype implementation of this construct in Berkeley UPC yields within 7% of ideal performance and 20-fold improvement over the original Standard UPC solution in some cases.

1 Introduction

In this paper, we explore the potential for asynchronous programming in a PGAS language, UPC [1], with MADNESS, an adaptively recursive algorithm operating on a distributed tree data structure. Our experience highlights the role of a new language feature, *asynchronous remote methods*, in providing substantial programmability and performance benefits to application developers. We prototype this feature in the Berkeley UPC implementation and report on its effectiveness. Our interest in asynchronous programming stems from a couple of factors:

1. Hardware trends dictate that users of high-end machines generate parallel work for progressively increasing processor counts with potentially different functional characteristics. Asynchronous programming is a powerful means

* This work has been supported by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory (ORNL), and the ORNL Postmasters Research Participation Program which is sponsored by ORNL and administered jointly by ORNL and by the Oak Ridge Institute for Science and Education (ORISE). ORNL is managed by UT-Battelle, LLC for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725. ORISE is managed by Oak Ridge Associated Universities for the U. S. Department of Energy under Contract No. DE-AC05-00OR22750.

of exposing the kinds of parallelism crucial to achieving portable high performance in this environment. Asynchronous computation is inherently oriented for greater computation-communication overlap and enables smarter runtimes performing dynamic load balancing, fault handling, and other resource management functions.
2. The application we are working with can be naturally written using asynchronous language constructs. Asynchronous programming is a convenient tool for developing an irregular code like ours that iterates recursively on a dynamically adaptive tree structure.

Our choice of PGAS, and UPC in particular, was motivated by:

1. PGAS is a convenient model for parallelizing an irregular, distributed application owing to its shared address space abstraction of the distributed execution environment and one-sided access mechanisms. While the newer HPCS languages are an attractive option for our application as they incorporate the PGAS model and are intrinsically geared towards asynchronous computing, they haven't yet attained full maturity [2]. More importantly, we wished to evaluate asynchronous programming in an SPMD context. And in early experience with MPI [3], two-sided message-passing communication proved to be a major programmability and performance limiter in expressing and managing the nested concurrency inherent in our application.
2. We were starting from a serial code comprising of non-trivial mathematical logic written in the C language. Going forward, we wished to avoid the significant effort required to translate this code into a different base serial language. As we explain later in the paper, the choice of UPC allowed us to retain and build upon this.

The rest of the paper is organized as follows. In Sect. 2, we describe the MADNESS application. In Sect. 3, we explain our solution in Standard UPC and discuss how it could be improved. We implement an asynchronous remote method extension to Berkeley UPC and discuss our modified solution in Sect. 4. We present related work in Sect. 5. We conclude and outline plans for continuing this work in Sect. 6.

2 The Problem

Multiresolution Analysis (MRA) is a mathematical technique for approximating a continuous function as a hierarchy of coefficients over a set of basis functions. MRA is characterized by dynamic adaptivity to guarantee the accuracy of approximation and being able to trade numerical accuracy for computation time. The natural representation of MRA is as a tree structure of coefficient tensors computed using recursive tree traversal algorithms. Multiresolution Adaptive Numerical Environment for Scientific Simulation (MADNESS) is a software package for multiresolution numerical methods on high-end computers, with application to numerous scientific disciplines. Its characteristics can be briefly summarized as follows:

1. Multi-dimensional tree distribution
   (a) The coefficient trees are unbalanced on account of the adaptive multiresolution properties leading to different scales of information granularity

Code 1 Serial Refinement - C

```
typedef struct {
  size_t order;
  GHashTable *tree;
} SubTree;
typedef SubTree *subtree_t;

typedef struct {
  subtree_t sumC;
  /* Math data structures */
} Func;
typedef Func *func_t;
```

```
void refine(func_t f,node_t node) {
  /* Math logic to calculate normf */

  for (int c=0;c<children(node);c++) {
    node_t child = get_child(node,c);
    if (normf > threshold)
      refine(f,child);
    else
      insert_coeffs(f,child);
  }
}
```

in different portions of the tree. Trees are multi-dimensional in nature i.e. binary tree in one dimension, quadtree in two dimensions, octree in three dimensions, and so on.

(b) The tree structure may be refined in an uncoordinated fashion – different parts of the tree may be refined independently and the intervals of such refinement are not preset.

(c) The tree partitioning scheme should cover the entire tree, and not just the leaf nodes, as a complete tree is utilized in some cases.

2. Algorithmic characteristics

(a) Some algorithms traverse the tree recursively, moving data up/down the hierarchy. The number of levels navigated varies dynamically and may constitute a data dependence chain.

(b) Some algorithms move data laterally within the same level in the tree i.e. between neighboring nodes.

(c) Some algorithms involve applying mathematical functions to the collection of coefficient tensors, and possibly combining individual results.

(d) Certain algorithms operate on multiple trees having different refinement characteristics and produce a new tree.

In this paper, we focus on the adaptive refinement that constructs the tree. Refinement proceeds in a top-down manner under the control of a threshold value that determines the precision of the numerical approximation. The operation starts at one node and progresses in a recursive way from parent to children nodes until the desired accuracy is achieved. Refinement may be initiated at any level in the tree, and may produce new nodes as the computation continues. An important feature is the flexibility to selectively refine sections of the tree as required. The irregularity arising from such flexible adaptive updates pose programmability and performance concerns in parallelization. Our starting point was a serial C version of the MRA refinement.

3 The Standard UPC Solution

We worked with two serial C codes: a one dimensional (1D) and a three dimensional (3D) refinement code. The two programs operate in an identical fashion, except that the underlying data structure is a binary tree in 1D and an octree in 3D. One other difference is that the computation is much more intensive in 3D than in 1D. Leaving aside these variations, the serial `refine` method is

Code 2 Asynchronous Refinement - UPC

```
typedef shared Func
 *GFunc_t;
typedef shared GFunc_t
 *gfunc_t;

gfunc_t gf=upc_all_alloc(
 THREADS,sizeof(GFunc_t));
gf[MYTHREAD]=upc_alloc(
 sizeof(Func));
```

```
void refine(gfunc_t gf,node_t node) {
  /* Math logic to calculate normf */

  for (int c=0;c<children(node);c++) {
    node_t child = get_child(node,c);
    if (normf > threshold)
      gtaskq_launch(gtaskq,thread_id,rtask);
    else
      gtaskq_launch(gtaskq,thread_id,ctask);
  }
}
```

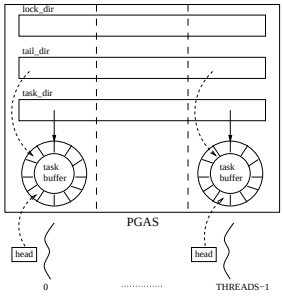
uniformly written as appears in Code 1. `func_t` defines the one/three dimension MRA function, with the corresponding tree form contained in `sumC`. The tree is structured as a hash table of (`node_t`,`tensor_t`) (definitions not shown) (key,value) pairs using the `GHashTable` data type from the GLib [4] utility library. `node_t` specifies node coordinates, while a coefficient tensor is of `tensor_t` type. The `insert_coeffs` method adds a (`node_t`,`tensor_t`) element to the MRA function tree when the `threshold` degree of precision is attained. A hash table relies on a hash function to uniquely identify its keys, in this case the node coordinates. Due to space constraints, we do not include a discussion of the derivation of node coordinates and the hashing technique, but mention that nodes are referenced by positional attributes in the tree hierarchy, which the hashing technique uses to differentiate one node from another.

3.1 Asynchronous Parallelization

The asynchronous solution creates *tasks* specifying *logical* parallelism for execution by physical UPC threads. In the refinement algorithm, operations on individual nodes can be formulated as separate parallel tasks. Starting as a single task to refine one node, the algorithm may recursively launch new tasks on the node hierarchy rooted at the initial node. A task is spawned on the thread that owns the target node; the identity of this thread is provided by a tree distribution scheme. A task will create child tasks if the numerical accuracy threshold mandates additional refinement of the tree structure beneath it, but does not wait on the execution of any spawned child tasks. A task always runs to completion and does not return control back to its parent. This process continues until all tasks on all threads have completed. We now discuss the various aspects of our parallel implementation.

Global data structures UPC's partitioned global address space offers a suitable mechanism to build our distributed shared data structures, see Code 2. `gfunc_t` is the globally shared MRA function, with each thread allocating a shared `Func` portion locally. `gfunc_t` is a portable global pointer that may be freely passed between threads and used by any thread to directly reference a remote section of the distributed MRA function.

Tree distribution A node is mapped to a UPC thread in a simple way by applying `(hash of node) modulo (thread count)`. This approach is characterized by poor locality arising from parent and children nodes being assigned




```
gtaskq_t gtaskq_init(size_t taskq_size,
 task_func_t *func_table,size_t ftable_size);

void gtaskq_launch(gtaskq_t gtaskq,
 size_t thread_id,task_t *task);

void gtaskq_execute(gtaskq_t gtaskq);

void gtaskq_destroy(gtaskq_t gtaskq);
```

Fig. 1. Global task queue - UPC

to different threads, but since it drives the placement of work in our owner-computes policy, it yields an excellent distribution of the workload.

Task queue A global task queue is at the heart of the asynchronous parallelization. Figure 1 depicts the task queue API and its key elements. Each thread allocates a shared bounded FIFO task buffer locally, and these are linked together in the global address space to constitute the global task structure `task_dir`. A distributed shared `tail_dir` tracks the next vacant position to insert a task in individual task buffers. Finally, a per-thread shared lock in `lock_dir` guards a thread's tail variable and task buffer from corruption in the face of concurrent task insertion attempts by multiple threads.

All threads call `gtaskq_init` with the `func_table` of methods in the program to run inside tasks for setting up the task queue. A thread deposits a `task` in the task buffer of thread `thread_id` by calling `gtaskq_launch`. All threads invoke `gtaskq_execute` to continually process tasks from their local task buffers until global termination is reached at which point all threads would have run out of tasks. `gtaskq_destroy` deallocates the different pieces of the task queue.

The task buffer is bounded by estimating the upper limit and hence we do not guard against buffer overruns. A more dynamic memory strategy will be necessary when the bounds cannot be accurately predicted. When a task executes, it may launch additional tasks as per the algorithm. To achieve termination of this dynamically unfolding task sequence, we applied the methodology of distributed termination without stopping work-related task communication [5]. We leveraged the same task launching and execution mechanisms from above to perform the various stages in the termination detection.

Asynchronous programming Putting together the global data structures, tree distribution and task queue, the parallel 1D and 3D `refine` methods are shown in Code 2. The `refine` method in Code 2 is passed the global MRA function `gf` and creates asynchronous tasks to run additional methods, but otherwise is identical to the `refine` method in Code 1. The computation begins with thread 0 launching a task to refine a given node on its host thread, and proceeds with all threads working collectively to execute the resulting dynamic set of tasks.

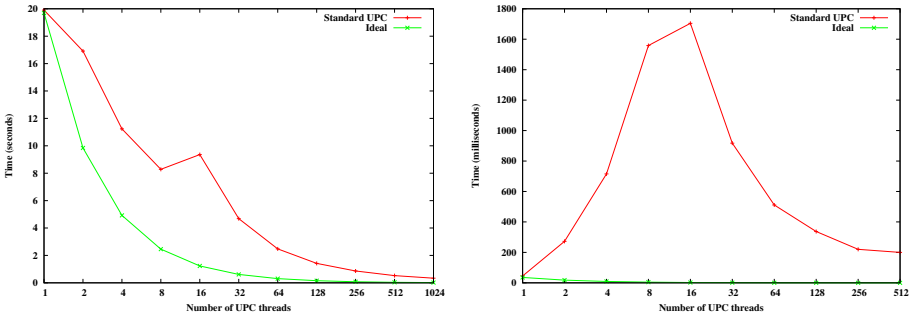


Fig. 2. Wall time of MADNESS in Standard UPC (a) 3D (b) 1D

3.2 Experimental Results

We ran the parallel codes at the National Center for Computational Sciences on the Smoky machine which is an 1280 core Linux cluster with 80 nodes, each node having four quad-core 2.0GHz AMD Opteron processors and 32 GB of memory, interconnected by InfiniBand network. The codes were compiled with Berkeley UPC version 2.8.0 to run on the OpenIB InfiniBand Verbs network conduit. We ran one UPC thread on one core, and each UPC thread comprised of two pthreads. The version of GLib for hash table support was 2.18.4.

We measured time on thread 0 from when it produces the first task to when it returns from the task execution loop on global termination for varying UPC thread counts. We define an *ideal* parallelization as the time for computing the same set of tasks without any of the associated parallelization overheads. To understand how we fared against the ideal parallelization scenario, we first obtained the execution time with a single thread after removing the locking/unlocking steps from the task launch and execute operations and privatizing access to the shared task queue. The ideal execution time for any given thread count was then derived by dividing this time with the particular number of threads. The results are plotted in Fig. 2. Not counting tasks that the task queue produces internally for achieving distributed termination, the 3D and 1D programs communicate 37440 and 6568 tasks respectively.

The 3D UPC solution is seen to be scaling reasonably well over the entire thread range, except for an increase in execution time from 8 to 16 threads, and exhibits better scaling beyond 16 threads. In contrast, the 1D code does not scale at all over the entire thread range. Its execution time continuously increases up to 16 threads, and scales only relatively thereafter. Both cases are characterized by significant loss in performance as measured by the difference between ideal and obtained run times.

3.3 Discussion of Results

To understand the observed trends, we gathered two sets of results. We provide the time for serial math processing inside the `refine` and `insert_coeffs` methods in Table 1. Next, we devised a micro-benchmark test to exercise the task launch and execute operations with a no-op method in the task. The test comprises of

Table 1. Computing time of MADNESS methods in microseconds

Code	refine	insert_coeffs
1D	3	1
3D	3099	153

Table 2. Task launch and execute times in microseconds

Count of threads	launch	lock	execute
No contention	81	20	0.43
2	91	25	91
4	313.99	200.33	106
8	676.43	567.86	97
16	1450.07	1336.07	97

running thread 0 as a receiver looping on its task buffer and executing incoming tasks. All other threads continually launch a specified number of tasks on thread 0. On receiving a task, thread 0 calls the no-op method, and returns to checking for new tasks. We note that this is the worst-case scenario in the solution we devised. We obtained the average time incurred in a task launch across the set of sending threads, as well as a breakdown of this time into its constituent parts. We also recorded the average time taken to execute one task on thread 0. The timings are captured in Table 2. Amongst the factors contributing to the launch time, only the time to acquire a remote lock is of interest here since other factors weren't seen to be contributing or varying greatly in comparison in different test scenarios.

The micro-benchmark reveals that even without competing threads, the time to launch a remote task far exceeds the processing within an individual task in 1D. The 3D tasks perform substantial work in comparison. With 2 threads, one sender and one receiver, the task execution time equals the time to launch a task. The no-op method in the task means both the sending and receiving threads are getting serialized on task communication. An increase in the number of sending threads is accompanied by a direct increase in the time to acquire a remote lock. The sending threads are all contending on the shared lock on thread 0. On obtaining the lock a thread inserts one task, but all other threads are left waiting on the completion of task insertions by one or more threads in the order that locks are acquired. Thus the latency experienced by a thread grows in proportion to its count of failed lock attempts, and the average task launch time steadily increases. Thread 0 is seen to take roughly the same amount of time to run one task since each successful task insertion implies one ready task for execution.

We are now in a position to interpret the graphs from Fig. 2. The overall scaling in 3D is due to the coarse-grained nature of its tasks that effectively mask task communication overheads. The extremely fine-grained tasks in 1D means its execution is dominated entirely by the cost of communication. The tree distribution scheme distributes the workload across the collection of threads, which means for greater thread counts the same set of tasks get farmed out to different threads, thereby reducing contention in task insertion. Consequently, scaling improves for larger numbers of threads. Still, the necessary serialization of task insertion is causing considerable performance degradation even in 3D as evidenced by the deviation from the ideal parallelization curve. Our hypothesis is we would address the performance problem if the task insertion could be spawned remotely and asynchronously. The UPC language specification does not have

constructs to support such asynchronous remote methods. We now implement this feature in Berkeley UPC to optimize our solution for performance.

4 Asynchronous Remote Methods (ARM)

An alternative approach to address the issues in the earlier solution is to maintain distinct task insertion slots for different threads and perform asynchronous insertions (using `bupc_memput_async`). Clearly, this method is not scalable, consumes memory, and is a programming burden. We instead choose to spawn the task launch contained in `gtaskq_launch` as an asynchronous remote method since it incurs minimal changes to the existing tasking infrastructure.

4.1 Implementation of Asynchronous Remote Methods

We describe a naïve prototype we developed to improve the performance of our code. It is not intended to be a proposal on how ARM could be enabled in Berkeley UPC.

Modifications to Berkeley UPC The Berkeley UPC runtime initially registers a set of function handles required by functionality such as locks and atomic updates with the GASNet [6] communication layer. To this list, we added several handles corresponding to our ARM implementations. To compile the modified Berkeley UPC source with our application, we provided dummy ARM definitions in a header file `upc_arm.h`. Along with the Berkeley UPC provided header file, this header file gets included in any UPC program. In the ARM version of the MADNESS UPC code, we selectively disabled the dummy ARM definitions and defined it by rewriting the task insertion from `gtaskq_launch` to be spawned as an active message. We added a construct `bupc_arm(arm_handler_index,thread_id ,buffer,size)` to Berkeley UPC to internally invoke the GASNet active message API with the task insertion function handle. `gtaskq_launch` now simply invokes `bupc_arm` to perform the task insertion as an ARM. The rest of the code base is identical to that of our earlier UPC solution.

ARM completion semantics We rely on the completion semantics provided by GASNet for the active message layer. We did not need to change the termination logic from before (same as in [5]). Despite the message ordering semantics of GASNet, on the single-rail InfiniBand network we used, all messages between pairs of processes are delivered in order. Hence, our ARM termination did not include message counters or similar logic for handling the unordered delivery of ARM messages. On systems where message ordering cannot be guaranteed, the termination would synchronize threads doing insertions and include information on the total number of messages.

Solution limitations Our ARM mechanism carries the restrictions of the GASNet active message API. We relied on the ordered delivery of messages and allowed only a static number of ARM definitions. All these limitations are specific to our solution.

4.2 Experimental Results and Discussion

With the same experimental setup from 3.2, we ran our MADNESS UPC program to insert tasks via ARM mechanism from within `gtaskq_launch`. The performance curves with this ARM execution along with the earlier results obtained

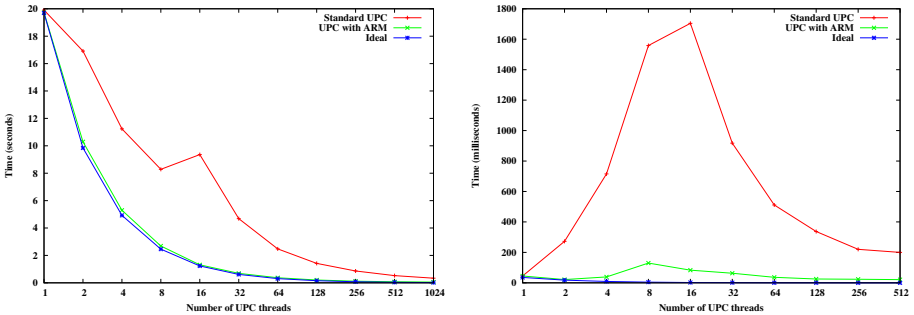


Fig. 3. Wall time of MADNESS in UPC with Asynchronous Remote Methods (a) 3D (b) 1D

using Standard UPC are shown in Fig. 3. The 3D case achieves very close to ideal performance and we are successful in overlapping most of its task distribution costs with task execution. Even 1D shows a clear performance gain, but its scaling is fundamentally limited by the amount of available computation. Benchmarking ARM reveals that task launching, while much less sensitive, still suffers a small penalty with multiple concurrent threads, consequently impacting scaling for smaller thread counts.

5 Related Work

The concept of asynchronous remote methods is not new to the HPC language community and appears in all the HPCS languages. For example, Chapel [7] provides statements like *begin*, *cobegin*, *coforall* and control over locality with an *on* clause for creating asynchronous parallelism on processing units or *locales*; X10 [8] has asynchronous constructs like *async*, *future*, *foreach*, *ateach* to specify parallelism on *places*. Co-Array Fortran [9], UPC [1], Titanium [10] language specifications do not explicitly support asynchronous programming, but their implementations utilize communication libraries that expose an active message interface. Scioto [11] offers a solution for efficient management of asynchronous parallelism on distributed memory machines. This work is different from all of the above since we evaluate asynchronous programming in an SPMD computation model from an application standpoint. While the dynamic load balancing problem of an asynchronous computation using standard UPC has been explored [12], we investigate how asynchronous programming could generally be made more effective in UPC with a new facility that allows greater asynchronicity in execution. The challenges of supporting dynamic parallelism in UPC were studied by applying UPC extended with user-level threads to dense LU factorization [13]. Our work is most similar to the work on programming a distributed spanning tree algorithm using LAPI's active message model [14]. But for greater ease of programming, we utilize a higher programming layer and demonstrate how performance could be improved without sacrificing programmability in this context via an extension built on top of GASNet's active message interface.

6 Conclusions and Future Work

We applied the asynchronous programming approach to parallelizing MADNESS, an adaptive algorithm that iterates recursively on tree data, in UPC. The shared view and independent remote memory access features of the PGAS model simplified the development of this dynamic and irregular application. By adding an asynchronous remote method construct, we achieved within 7% of ideal performance and 20-fold improvement over the Standard UPC solution in some cases. Though this is a problem-specific prototype, we demonstrated that asynchronous remote methods can provide substantial programmability and performance benefits to application developers. We hope that our work motivates the inclusion of such a capability in the UPC standard. As future work, we plan to expand the design of asynchronous remote methods to be more general, portable, and adaptable to different UPC applications. We would like to demonstrate its scaling on bigger machines. We intend to implement more of the MADNESS application kernels in UPC to derive a general solution for use in benchmarking UPC implementations and systems that are suitable for the APGAS model.

Acknowledgements

We wish to thank Brian Larkins from The Ohio State University for providing the serial C version of the MRA refinement program used in this work.

References

1. UPC Consortium: UPC Specification, v1.2. Technical Report LBNL-59208
2. Shet, A., Elwasif, W., Harrison, R., Bernholdt, D.: Programmability of the HPCS languages: A case study with a quantum chemistry kernel. In: IPDPS 2008. 1–8
3. Message Passing Interface Forum: MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`
4. GNU Project: GLib. `http://library.gnome.org/devel/glib`
5. Francez, N., Rodeh, M.: Achieving distributed termination without freezing. Software Engineering, IEEE Transactions on **SE-8**(3) (May 1982) 287–292
6. Bonachea, D.: GASNet Specification, v1.1. Technical Report CSD-02-1207
7. Cray Inc.: Chapel Language Specification, v0.782. `http://chapel.cs.washington.edu/spec-0.782.pdf` (April 2009)
8. IBM: Report on the Experimental Language X10, v1.7. `http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf` (March 2009)
9. Numrich, R., Reid, J.: Co-Array Fortran for parallel programming. ACM Fortran Forum **17**(2) (1998) 1–31
10. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance java dialect. In: ACM. (1998) 10–11
11. Dinan, J., Krishnamoorthy, S., Brian, L., Nieplocha, J., Sadayappan, P.: Scioto: A framework for global-view task parallelism. In: ICPP 2008. 586–593
12. Olivier, S., Prins, J.: Scalable dynamic load balancing using UPC. In: ICPP 2008. 123–131
13. Husbands, P., Yelick, K.: Multi-threading and one-sided communication in parallel LU factorization. In: SC 2007. 1–10
14. Cong, G., Xue, H.: A scalable, asynchronous spanning tree algorithm on a cluster of smps. In: IPDPS 2008. 1–6