2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

Asynchronous Work Stealing on Distributed Memory Systems

Shigang Li[1], Jingyuan Hu[2], Xin Cheng[1], Chongchong Zhao[1]
[1]School of Computer and Communication Engineering, University of Science and Technology Beijing, China
[2]Universit´e de Technologie de Troyes, France
lsgwoody@gmail.com, jingyuan.hu@utt.fr, chengxin0613@gmail.com, zhaochongchong@tsinghua.org.cn

***Abstract*—Work stealing is a popular policy for dynamic load balancing of irregular applications. However, communication overhead incurred by work stealing may make it less efficient, especially on distributed memory systems. In this work we propose an asynchronous work stealing (AsynchWS) strategy which exploits opportunities to overlap communication with local residual tasks. Profiling information is collected locally to optimize task granularity and guide the asynchronous work stealing. AsynchWS is implemented in Unified Parallel C (UPC), which effectively supports non-blocking one-sided communication and facilitates the implementation. Experiments are conducted on a 32 nodes Xeon X5650 cluster using a set of irregular applications. Results show that up to 16% better performance than the state-of-the-art strategies on distributed memory.**

Keywords-asynchronous work stealing; distributed memory; task granularity; UPC

I. INTRODUCTION

Dynamic task parallelism aims at efficiently parallelizing irregular constructs, such as while loops and recursive structures, in which the work and parallelism unfold throughout the program execution. Languages or libraries are developed to support dynamic task parallelism, for instance, Cilk [5], Intel Threading Building Blocks (TBB) [14], Java's Fork-join Framework [15], Microsoft Task Parallel Library [17], OpenMP 3.0 [18] on shared memory system, and Asynchronous Dynamic Load Balancing (ADLB) [6], Charm++ [16], task library in PGAS languages [3, 10, 19] on distributed memory. Research work mainly focuses on task creation [9, 11, 12], task scheduling [8, 13] and the locality of work-stealing [2, 3, 13].

This paper introduces AsynchWS, an asynchronous work stealing strategy for distributed memory systems. Dynamic load balancing is essential for the performance of irregular applications. However, communication overhead incurred by work stealing may make it less efficient. AsynchWS extends state-of-the-art techniques [10, 3] to support asynchronous work stealing for both stealer and victim threads. AsynchWS adopts a dynamic cut-off strategy to reduce task creation overhead and increase the task granularity. Profiling information is collected locally, i.e. within node, to estimate the task grain size and guide the asynchronous work stealing. AsynchWS is implemented in a PGAS language, UPC [1]. UPC features global array which brings us a straightforward method to build distributed task queue structure. One-sided communication in UPC, which decouples data transfer from processor synchronization, facilitates communication-computation overlap. We evaluate the AsynchWS strategy with irregular applications on a 32 nodes Xeon X5650 cluster. Experimental results show up to 16% better performance than the state-of-the-art strategies.

The paper is organized as follows: Section 2 introduces the basic structures to support task parallelism on distributed memory. Section 3 presents an adaptive cut-off strategy. Section 4 shows the implementation of asynchronous work stealing strategy. Experimental results and analysis are presented in Section 5. Section 6 and Section 7 presents related work and conclusions respectively.

II. TASK PARALLELISM ON DISTRIBUTED MEMORY

Previous work [10, 3] discussed the implementation of work stealing in large scale systems using PGAS language. We utilize the basic structures in their work, such as distributed task queue, hierarchical victim selection, task dependency and termination detection, to build the framework that supports task parallelism on distributed memory.

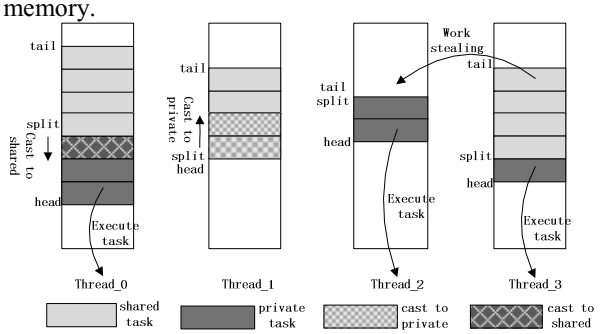

Figure 1. Distributed task queue

A global task queue is distributed among all the UPC threads as illustrated in Fig. 1. Each thread maintains its own task queue divided into a private region and a shared region. Tasks in the private region can be executed directly by the related thread while tasks in the shared region can be stolen by other threads. When a task is created by a thread, it is firstly put in the head of the private region. When the number of private tasks exceeds the threshold, the excess portion will be casted to shared tasks by updating the split pointer, such as the status of Thread_0. When a thread finishes all the private tasks and still has shared tasks in its task queue, it will cast a portion or all the shared tasks to private, such as the status of Thread_1. When a thread has no private and shared tasks, it will choose a victim and try to steal tasks from the tail of victim task queue. Actually in AsynchWS,

1066-6192/12 $26.00 © 2012 IEEE
DOI 10.1109/PDP.2013.35

work stealing may be triggered when there are still a small number of residual tasks, such as the status of Thread_2 and Thread_3. The distributed task queue structure and separate private and shared regions minimize the contention of task scheduling.

III. ADAPTIVE CUT-OFF STRATEGY

A. Adaptive Cut-off

Creating too many fine grain tasks always leads to inefficiency. In AsynchWS, we use an adaptive cut-off strategy which aims at creating tasks according to its estimated work load. We use help-first scheduling policy [8] at the beginning, namely threads execute the continuation and leave the spawned task to be stolen, to spawn tasks quickly, and then switch to work-first scheduling policy [8], namely threads execute the spawned task eagerly and leave the continuation to be stolen, to sample the runtime of executed tasks. Work load of one task is estimated as the sum of child-task work load. Average work load of sampled tasks in one depth level is used as the estimated grain size for each task in this level. Typically, a task is worth to create when the work load is more than a few thousand mathematic calculations [14]. Currently the order of magnitude of frequency in modern processors is GHz, so the reasonable task grain should be several milliseconds runtime. In our strategy, the minimum task grain is set as 2 milliseconds. In order to reduce profiling overhead, we set the maximum number of sampled tasks for a given depth level as $\min(2^n / 100, 16)$, where n is the depth level. The algorithm of adaptive cut-off strategy is presented in Fig. 2.

Configuration: task queue bound = 3*threads number in a node; number of samples at each depth level <= 16 in a node; minimum task grain = 2 milliseconds.
1. All threads, at the beginning, adopt help-first scheduling policy until reach the 2/3 task queue bound.
2. All threads switch to work-first scheduling policy to gather profiling information, namely the work load. If enough tasks have been sampled, profiling at this level is closed. Average work load of sampled tasks is used as the estimated grain size for each task in this level.
3. Each thread determines to create task or not according to the estimated task grain and task queue bound.

```
if task grain is not estimated then
    if ready tasks have not reached the 2/3 task queue
       bound then create task;
    else not create task.
else if the estimated task grain > minimum task grain
       and tasks have not reached task queue bound
    then create task;
    else not create task.
```

Figure 2. Algorithm of adaptive cut-off

This strategy improves bounded task queue cut-off method by avoiding fine grain tasks. Fig. 3 shows how it works for an unbalanced tree. After reaching 2/3 task queue bound, each thread switches to work-first scheduling to get profiling information. In Level 5, the estimated grain size is less than the minimum size, and then no task is created at this level. However, task 7 has coarse grain which should be spawned. This can be compensated by creating tasks at Level 6 and Level 7. Level 8 does not create tasks for the same reason as Level 5.

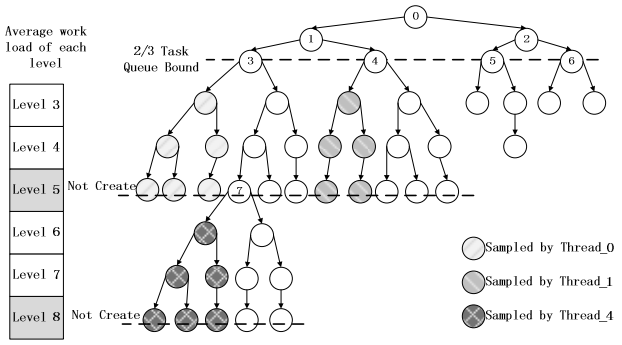

Figure 3. Adaptive cut-off for an unbalanced tree, 4 participating threads

B. Profiling Information Collection

During collecting the profiling information, mutual exclusion is needed when accessing the shared data structure, namely the average work load of each level presented in Fig. 3. Because tasks may be stolen by other threads, one task adding its runtime to the runtime of its parent task should be protected by mutual exclusion. Also, task updating the profiling information of the corresponding tree depth level should be mutually exclusive. However, collecting profiling information across nodes (global profiling) is costly. In our strategy, profiling information is only collected within a node (local profiling), and each thread uses the local information to guide task creation. When an inter-node work-stealing happens, if the estimated task grain is more than 4 milliseconds or the task grain has not been estimated, the stolen task will be sampled in the domain of the stealer; otherwise it will simply serialize the child tasks.

IV. ASYNCHRONOUS WORK STEALING

A. Overlapping Communication with Residual Work Load

AsynchWS tries to find out the opportunity to overlap communication with residual work load, namely stealing tasks in advance when there is no task in shared region but still some residual tasks in the private region. In order to implement asynchronous work stealing, we use the Berkeley UPC extension function **bupc_memget_async()** to steal tasks from victims. It is a non-blocking point-to-point communication operation and the function returns a handle. The handle is used by **bupc_waitsync()** which is a blocking wait. During calling **bupc_memget_async()** and **bupc_waitsync()**, residual tasks are executed. The framework of AsynchWS is presented in Fig. 4. Note that two gray boxes are overlapped and neither the stealer nor the victim is blocked when work stealing happens. There is a trade-off when determining the number of residual tasks that triggers work stealing. If the number is too small, there is probably not sufficient available computation to do overlap.

However, if the number is too large, it may damage the load balancing.

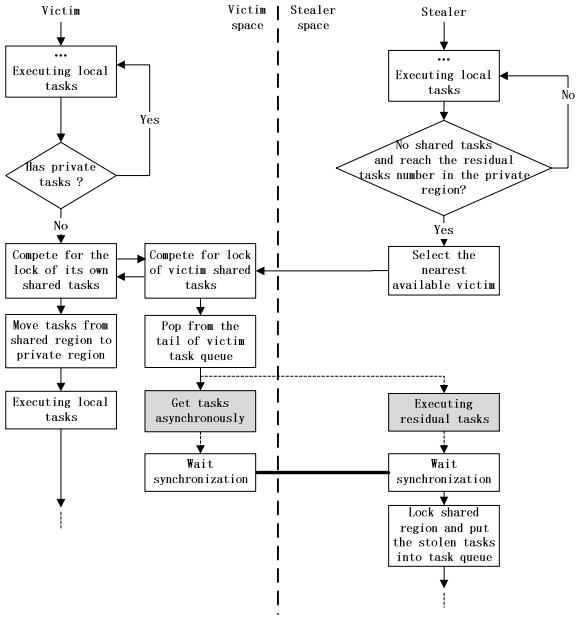

Figure 4. Asynchronous work stealing protocol

B. Number of Residual Tasks

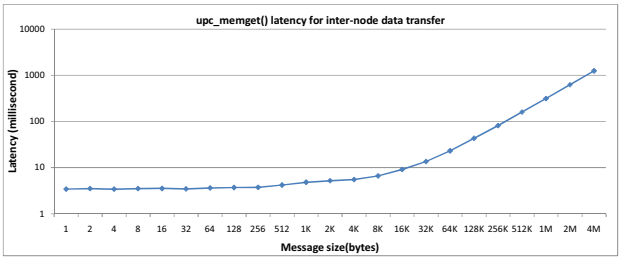

Figure 5. Latency of upc_memget() for inter-node data transfer

In order to distribute tasks quickly on distributed memory, steal-half strategy is used in AsynchWS, namely a stealer steals half of shared tasks from a victim. Typically data size transferred from victim to stealer, including the basic information and the input data of a task, is about 120 bytes, as shown in Table 1. In AsynchWS, we set the bound of shared tasks of each thread as 24. So when using steal-half strategy, the data size approximately varies from 120 bytes to 1.4K bytes. We test **upc_memget()** latency for inter-node data transfer on Xeon X5650 cluster connected by Voltaire QDR Infiniband, and the results are presented in Fig. 5. The latency of inter-node work stealing varies from 3.7 milliseconds to 5.2 milliseconds according to the experimental results. Recall that we set the minimum task grain as 2 milliseconds, so 2 or 3 residual tasks in the private region are used to overlap the communication. In order to avoid leaving coarse grain residual tasks, profiling information are used to guide work stealing. In our strategy, if the total estimated task grain size of the residual tasks is larger than 12 milliseconds, the thread will continue to execute the next task, otherwise asynchronous work stealing is triggered.

V. EVALUATION

A. Experimental Setup

Experiments are conducted on 32 nodes (total 384 cores) Xeon X5650 cluster connected with Voltaire QDR Infiniband. Each node contains two 2.67 GHz Intel Xeon hex-core processors. Operating system is Scientific Linux 6.1.The version of Berkeley runtime and UPC-to-C translator is 2.14.0. The back-end compiler is gcc 4.4.5, with its -O3 option turned on. We use six kernels from Barcelona OpenMP Task Suite [4] to evaluate the AsynchWS strategy: Fibonacci, Floorplan, NQueens, SparseLU, Strassen and UTS. Scale of input data and data size of one task are presented in Table 1 for each application.

TABLE I. PARAMETERS FOR APPLICATIONS

Kernels	Parameters	
	Input	***Data Size of a Task***
Fibonacci	n=50	92 bytes
Floorplan	20 Cells	160 bytes
NQueens	14*14 chessboard	164 bytes
SparseLU	4096*4096 matrix	108 bytes averagely
Strassen	8192*8192 matrix	116 bytes
UTS	T3L	120 bytes

B. Overhead of Profiling

Overhead of profiling mainly comes from the contention of accessing shared data structure. However, testing the overhead of profiling in the irregular applications is difficult because it mixes with cut-off strategy. In this experiment, we simply use the depth of tasks tree of each kernel and the number of samples at each level to simulate the profiling of overhead. For local profiling, the number of samples at each depth level is designated less than 16 while global profiling has the same total number of samples. From Fig. 6, we can see the overhead of local profiling is less than 2% of the total runtime for all six applications, but global profiling is much higher, 23.6% averagely.

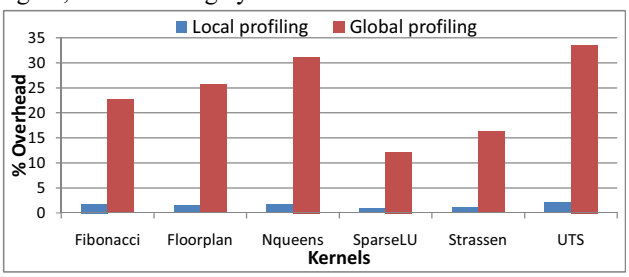

Figure 6. Profiling overhead, total 384 threads

C. *Impact of Residual Tasks Number*

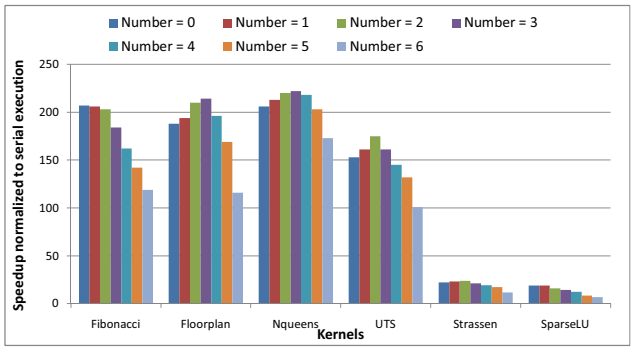

Figure 7. Speedup of different residual tasks number, total 384 threads

The performance of AsynchWS is sensitive to the number of residual tasks in the private region used to overlap communication. The golden point is the workload of residual tasks equals to the communication overhead. We compare the performance of residual tasks number from 0 to 6 and the results is presented in Fig. 7. Fibonacci gets the best performance when there is no residual task and number equaling 1 or 2 is a little worse than the best performance. This is because the task tree of Fibonacci is very balanced and there are few steals. The benefit of overlapping cannot amortize work imbalance. Floorplan, Nqueens and UTS get the best performance when the number equals 2 or 3 just as we have analyzed in Section 4. The task tree for these three kernels is imbalanced which leads to more steals, so they get apparent performance improvement from AsynchWS. Strassen get the optimal performance when the number equals 2. However, SparesLU gets the best performance when there is no residual task. This is because SparseLU has coarse grain tasks, so even small number of residual tasks can lead to load imbalance. In the following experiments, the residual tasks number is configured to 2 because most applications can get the optimal or sub-optimal performance.

D. *Impact of Different Cut-off Strategies*

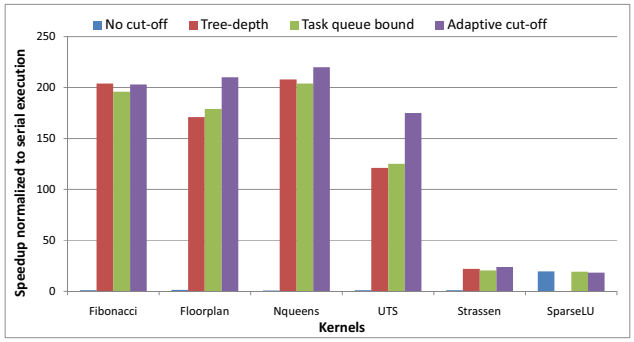

Figure 8. Speedup of AsynchWS combined with different cut-off strategies, total 384 threads. Tree-depth is the best version tree-depth cut-off tuned manually. Task queue bound means task creation is bounded by the number of task in task queue. Adaptive cut-off is the strategy we presented in Section 3

We also test the performance when AsynchWS is combined with different cut-off strategies and the results are shown in Fig. 8. When there is no cut-off all the applications except for SparseLU get speedup less than 1.4. This is caused by the high overhead of fine grain task creation. SparseLU has parallel-for parallelism with no nested tasks and task grain is coarse, so it cannot use tree-depth based cut-off and other three strategies get similar speedup. Floorplan, Nqueens, UTS and Strassen get the best performance when AsynchWS is combined with the adaptive cut-off because profiling information can guide AsynchWS to avoid large grain residual tasks. However, other strategies can lead serious to load imbalance especially for unbalanced tasks tree, such as Floorplan and UTS. Fibonacci gets the best performance using tree-depth based cut-off while adaptive cut-off is a little worse.

E. *Compare performance of AsynchWS with synchronous work stealing (SynchWS)*

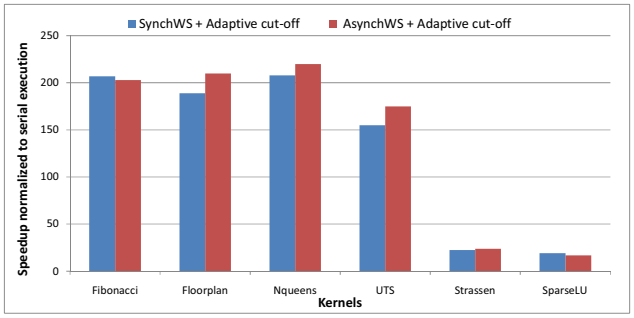

Figure 9. SynchWS vs AsynchWS, both combined with adaptive cut-off

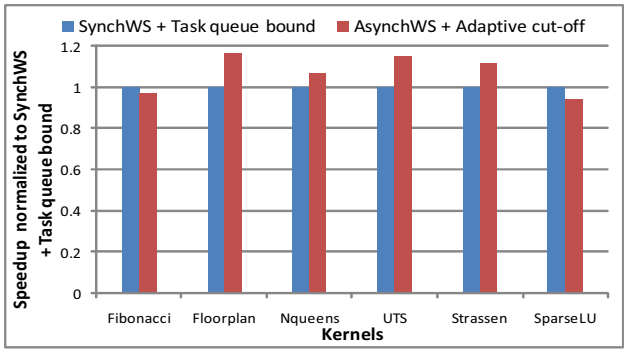

Figure 10. SynchWS vs AsynchWS, combined with task queue bound and adaptive cut-off respectively

Fig. 9 compares the performance of SynchWS and AsynchWS. SynchWS means stealer has to wait for the stolen tasks but victim is not blocked. AsynchWS is the protocol presented in Section 4 and both stealer and victim are not blocked. We can see that AsynchWS works apparently better for the unbalanced tasks trees which have more work stealing, such as Floorplan, UTS and Nqueens. SynchWS is better than AsynchWS in SparseLU because coarse grain tasks in SparseLU lead to load imbalance when using AsynchWS. There is no apparent performance difference in Fibnacci and Strassen because less work stealing is triggered. Fig. 10 compares the performance of AsynchWS combined with adaptive cut-off and SynchWS

combined with task queue bound, which is used in HotSLAW, and AsynchWS shows 16%, 6%, 14% and 11% better performance for Floorplan, Nqueens, UTS and Strassen while SparseLU and Fibonacci are a little worse.

VI. Related Work

Dynamic task parallelism and work stealing have been widely studied in the literature. SLAW [13] and HotSLAW [3] focus on the locality of work stealing. SLAW extends work stealing for shared memory with a notion of locality. HotSLAW steps further to address the hierarchical nature of memory locality on distributed memory systems and tries to steal work from the lowest hierarchy level. This strategy is also used in AsynchWS to protect locality on distributed memory.

Some research work has been done in the area of scalable work-stealing. Olivier and Prins [19] present an optimized runtime library of work stealing for the UTS benchmark in UPC. Dian et.al [10] present a scalable implementation of work stealing using PGAS programming model provided by ARMCI on multi-core clusters. Chase and Lev [7] presents a lock-free work-stealing deque, which can dynamically grow when it overflows.

For the task granularity, Tascell [9] proposes a back-tracked scheduling approach in which the program runs sequentially and back-tracks when there is a steal request. Tzannes et.al [11] have implemented a scheme that decides at run-time whether to further sub-divide and spawn the iteration range based on the current status of the task queue. Duran et.al [12] proposed an adaptive cut-off technique for OpenMP tasks applications on shared memory. Profiling information is collected to estimate task grain size. This is similar to the adaptive cut-off in AsynchWS, but AsynchWS uses the number of in task queue as the bound of task creation, which can void unreasonably serializing a large sub-tree too soon.

VII. Conclusions and Future Work

We present an asynchronous work stealing (AsynchWS) strategy on distributed memory systems. AsynchWS aims at finding opportunities to overlap communication with local residual tasks, in which neither the stealer nor the victim is blocked when work stealing happens. Beside, profiling information is collected locally to optimize task granularity and guide the asynchronous work stealing. Experimental results show that up to 16% better performance than the state-of-the-art strategies. AsynchWS can get better performance for unbalanced, fine grain tasks applications. However, for balanced or coarse grain tasks applications, AsynchWS cannot get better performance because of leading to load imbalance. To solve this problem is our future work.

Acknowledgment

The research is partially supported by the Key Project of the National Twelfth-Five Year Research Program of China under Grant No.2011BAK08B04, Key Project of the National Science and Technology of China under Grant No.2009ZX03004-004, Hi-Tech Research and Development Program (863) of China under Grant No. 2008AA01Z109.

References

[1] "UPC language specifications, v. 1.2," Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures, 2000, pp. 1–12.

[3] S. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models, 2011.

[4] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguad´e, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in 38th International Conference on Parallel Processing, Vienna, Austria, IEEE Computer Society, 2009, pp. 124-131.

[5] 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, New York, NY, USA, ACM, 1998, pp. 212–223.

[6] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," SciDAC Review, vol. 17, 2009, pp. 30–37.

[7] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures, 2005.

[8] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in IPDPS'09, 2009, pp. 1–12.

[9] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based load balancing," in Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. New York, NY, USA: ACM, 2009, pp. 55–64.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable Work Stealing," in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC'09, New York, NY, USA, 2009, pp. 1-11.

[11] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary splitting: A run-time adaptive dynamic works-stealing scheduler," in Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming, ACM Press, New York, 2010, pp. 179–190.

[12] A. Duran, J. Corbalan, and E. Ayguad ´ e, "An Adaptive Cut-off for Task Parallelism," in Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008.

[13] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," In IPDPS'10, 2010, pp. 1-12.

[14] "Intel(R) Threading Building Blocks," Intel Corporation, 2007.

[15] D. Lea, "A java fork/join framework," in Proceedings of the ACM 2000 conference on Java Grande, New York, NY, USA: ACM, 2000, pp. 36–43.

[16] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, 1993, pp. 91-108.

[17] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," In OOPSL'09, 2009, pp. 227-242.

[18] "OpenMP Application Program Interface, v. 3.0," OpenMP Architecture Review Board, 2008.

[19] S. Olivier and J. Prins, "Scalable Dynamic Load Balancing Using UPC," in Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA, 2008, pp. 123–131.