Hierarchical Work Stealing on Manycore Clusters

Seung-Jai Min[1] Costin Iancu[1] Katherine Yelick[1,2]

[1]Lawrence Berkeley National Laboratory and [2]University of California at Berkeley

Abstract

Partitioned Global Address Space languages like UPC offer a convenient way of expressing large shared data structures, especially for irregular structures that require asynchronous random access. But the static SPMD parallelism model of UPC does not support divide and conquer parallelism or other forms of dynamic parallelism. We introduce a dynamic tasking library for UPC that provides a simple and effective way of adding task parallelism to SPMD programs. The task library, called HotSLAW*, provides a high-level API that abstracts concurrent task management details and performs dynamic load balancing. To achieve scalability, we propose a topology-aware hierarchical work stealing strategy that exploits locality in distributed-memory clusters. Our approach, named* HotSLAW*, extends state of the art techniques in shared- and distributed-memory implementations with two mechanisms: Hierarchical Victim Selection (HVS) finds the nearest victim thread to preserve locality and Hierarchical Chunk Selection (HCS) dynamically determines the amount of work to steal based on the locality of the victim thread. We evaluate the performance of our runtime on shared- and distributed-memory systems using irregular applications. On shared memory,* HotSLAW *provides performance comparable or better than hand tuned OpenMP implementations. On distributed memory systems, the combination of Hierarchical Victim Selection and Hierarchical Chunk Selection provides better performance than state of the art approaches using a random victim selection with a StealHalf strategy for the workload considered.*

1. Introduction

The Unified Parallel C (UPC) [22] language provides the convenience of a global address space with the locality control needed for high performance and scalability. It has been shown to perform well on both shared and distributed memory machines, and the one-sided communication model that results from reading and writing to the global address space is both lightweight and takes advantage of modern interconnect features. However, UPC uses a Single Program Multiple Data (SPMD) model of parallelism in which a fixed set of threads run throughout the program execution. While that SPMD model matches the underlying hardware, it does not directly support applications that involve dynamic tasking. Dynamic tasking is especially important for problems in which the work and parallelism unfold dynamically throughout the program execution. In this case, it is impossible to determine *a priori* how to divide work among processors so as to evenly balance the load. With dynamic tasking, computations are explicitly and dynamically created and a runtime scheduler provides load balancing services:work-stealing is the most popular runtime scheduling approach.

In this paper we present a dynamic tasking library called *HotSLAW* for UPC. HotSLAW provides a high-level API that abstracts concurrent task management details and performs dynamic load balancing. Besides API expressiveness and conciseness, a major goal of our library is to achieve scalable and robust performance on distributed memory multicore clusters. The implementation takes advantage of the one-sided data access mechanism to implement work-stealing efficiently on large scale systems, and also builds on prior art in dynamic load balancing for both shared and distributed memory machines. This includes a large body of research on dynamic tasking and work-stealing on shared memory architectures, such as Cilk [8], Intel Threading Building Blocks, Microsoft Task Parallel Library, and OpenMP 3.0. Distributed memory tasking at scale has been demonstrated using one-sided communication paradigms such as the Aggregate Remote Memory Copy Interface [14] (ARMCI) [3, 4] or Unified Parallel C [17, 21].

Our work builds in particular on that of Guo et al [10] who show that scalable work stealing on shared memory systems is achieved using a dynamic task scheduling policy with locality awareness. Their Scalable Locality-aware Adaptive Work-stealing scheduler (SLAW) combines work-first and help-first scheduling policies while ensuring bounded memory usage. HotSLAW extends the principles behind these state-of-the-art approaches to address the inherently hierarchical nature of memory locality in modern systems, without assuming application or language level locality hints. SLAW allows stealing to occur only within a *place* (a programmer defined locality domain), while HotSLAW provides a hardware topology-aware *hierarchical victim selection* strategy where the algorithm tries to steal work from the lowest hierarchy level (e.g. socket) before moving up the hierarchy (e.g. inter-socket or inter-node). For distributed memory, we extend the implementation proposed by Dinan et al [4]. SLAW steals a fixed amount of work and Dinan's implementation steals a fixed ratio of work per event. HotSLAW uses a *hierarchical chunk selection* approach to dynamically select the amount of work stolen based on the "locality" hierarchy used.

We have evaluated our approach using programs with irregular parallelism on shared and distributed memory systems up to 256 cores. The benchmarks and the experimental setup are described in Section 4. On shared memory systems, HotSLAW is able to match or exceed by as much as 109% the performance of manually optimized OpenMP implementations. On distributed memory systems, hierarchical victim selection is able to improve performance by up to 52% when compared to the default random victim selection policy. Enabling hierarchical chunk selection provides performance improvements up to 122% when compared to the *StealHalf* method for distributed memory introduced by Dinan et al [4]. Our experimental results indicate that using a three level (intra-, inter-socket and inter-node) hierarchy, HotSLAW is able to match hand tuned performance obtained using an exhaustive search of the space of the parameters that determine the performance of

work-stealing: scheduling policy, space bounds, victim selection, and stealing granularity.

The rest of the paper is organized as follows. We describe the task parallel programming API for UPC in Section 2. Section 3 explains the design and implementation of HotSLAW. We present the experimental setup in Section 4 and a thorough performance evaluation of HotSLAW in Section 5. We discuss related work in Section 6, followed by a conclusion in Section 7.

2. Unified Parallel C

Unified Parallel C (UPC) is an extension to ISO C 99 that provides a Partitioned Global Address Space (PGAS) abstraction using Single Program Multiple Data (SPMD) parallelism. The memory is partitioned in a task local heap and a global heap. All tasks can access memory residing in the global heap, while access to the local heap is allowed only for the owner. The global heap is logically partitioned between tasks and each task is said to have local affinity with its sub-partition. Global memory can be accessed either using pointer dereferences (load and store) or using bulk communication primitives (`memget()`, `memput()`). The language provides synchronization primitives, namely locks, barriers and split phase barriers. Most of the existing UPC implementations also provide non-blocking communication primitives, e.g. `upc_memget_nb()`. The language also provides a memory consistency model which imposes constraints on message ordering.

2.1 UPC Task Library API

```
taskq_t *taskq_all_alloc(int nFunc, void *func1,
        int input_size1, int output_size1, ...);
int taskq_put(taskq_t *taskq, void *func,
        void *in, void *out);
int taskq_execute(taskq_t *taskq);
int taskq_steal(taskq_t *taskq);
void taskq_wait(taskq_t *taskq);
void taskq_fence(taskq_t *taskq);
int taskq_all_isEmpty(taskq_t *taskq);
```

Figure 1. Task Library API

We provide a library API for instantiating and controlling dynamic task parallelism from UPC applications. Figure 1 shows a list of the most commonly invoked functions. The `taskq_all_alloc` function allocates a distributed data structure to represent a global task queue; this is a collective function call where arguments have to match across all threads. The first input argument, *nFunc* is the number of different task functions that can be put into this task queue. Each function is represented by a triplet which consists of a pointer to the proper function, an input data size, and an output data size. Logically, the task queue is split into a thread private portion (or local) and a public portion.

The `taskq_put` function creates a task and puts it into the queue when space is available. As described later, we implement a combination of the work-first and help-first scheduling policies with bounded queues and tasks can be inlined (serialized) when space is unavailable.

The `taskq_execute` function removes a task from the head of the private task queue and executes it, while the `taskq_steal` function attempts to steal tasks from the public queue. We further discuss the stealing strategy in Section 3.2.

The library also provides primitives for inter-task synchronization and ordering. The `taskq_fence` is a non-blocking operation used to enforce ordering between task sub-graphs: it ensures that any task spawned after calling fence will not be scheduled for execution until all the tasks spawned before the fence have completed. The `taskq_wait` is a blocking call that terminates when all the spawned tasks are completed. The `taskq_wait` function internally calls the `execute` and `steal` functions to ensure progress.

The `taskq_all_isEmpty` function is a collective function for termination detection, it returns 1 if the global task queue is empty, otherwise it returns 0. In addition, we provide configuration functions `taskq_set_*` to specify user defined stealing hierarchies and configurable behavior.

The complete description of the task library API can be found at `http://upc.lbl.gov/task`.

2.2 Programming Example

```
01: void FIB( int *n, int *out ) {
02:   int n1 = *n-1;
03:   int n2 = *n-2;
04:   int x, y;
05:   if (*n < CUTOFF) {
06:     FIB_serial(n, out);
07:     return;
08:   }
09:   taskq_put(taskq, FIB, &n1, &x);
10:   taskq_put(taskq, FIB, &n2, &y);
11:   taskq_wait(taskq);
12:   *out = x + y;
13: }
```

Figure 2. UPC task example for Fibonacci

Recursive divide-and-conquer functions and parallel-for loops (a.k.a. parallel do-all loops) with potential load imbalance are good candidates for dynamic tasking. Figure 2 shows a Fibonacci numbers generator implemented using our API. A task function `FIB` spawns two children tasks at lines 9 and 10 and waits at line 11 until these children complete. While waiting inside the `taskq_wait` function, the runtime library will consume tasks or try stealing from other threads if the local task queue is empty.

Tasks are specified at function granularities, with a signature containing pointers to input and output data buffers, e.g.:

```
void my_func(void *input, void *output);
```

In contrast with the API used by Dinan [4] which uses global addresses to specify input/output data, we use proper C `void*` pointers in order to improve interoperability with other programming models. Input and output data are stored in contiguous memory space. Input data is copied into the library space when a task is created and travels with it whenever migration occurs.

To exploit the PGAS support of UPC, data fields in the input and output regions can be either a value or a reference. However, if such a data is a reference type, it should be a pointer to shared data which remains valid after steals. A task function can read its local variables, its input parameter, shared variables, and file scope constant variables and can write to its local variables, its output parameter, and shared variables.

Our library maintains and at termination propagates the content of the task output buffer. The content of this region is undefined until task termination.

3. HotSLAW Implementation

Work-stealing has been widely popularized by Cilk [8] and extensively studied. Implementations use two scheduling policies: work-first and help-first. Under work-first, the scheduler executes the spawned task eagerly and leaves the parent task to be stolen. Since we provide a library based approach, in our context a work-first policy amounts to invoking the task function directly from the

`taskq_put` library call. With the help-first policy, new tasks are made available for stealing while the execution continues with the parent task.

It is reported [9, 10] that the work-first policy is good for scenarios when stealing is rare, but its implementation can cause stack overflow for applications with very deep task execution trees, while the help-first policy is favorable when stealing is frequent and can be implemented with low stack usage but is not space-efficient in general. As any work-first implementation, ours too has the potential for stack overflow.

SLAW [10] presents an implementation using an adaptive policy: tasks are generated and initially executed using a help-first policy until several constraints are met, at which point the scheduler switches to a work-first policy. In order to reduce overhead, SLAW re-evaluates the spawning policy after a "certain" number of spawn operations. In our implementation, we use the SLAW approach with simpler heuristics to implement queue bounds and policy switches.

SLAW use three parameters to guide policy selection: S represents a threshold for the number of active stack frames, after which the implementation switches from work-first to help-first; F is the threshold on number of tasks in the queue after which the implementation switches from help-first to work-first; and INT is the periodical number of events after which the scheduling policy is re-evaluated. A typical execution starts with help-first to build parallelism and then switches to work-first in order to reduce the overhead of task creation and execution. If the stack bound S is reached, the execution reverts to help-first and increases heap memory pressure.

In our implementation we use bounded queues, which are the equivalent of the parameter F in SLAW. The execution starts with help-first, and when queue space is exhausted, it reverts to work-first. Help-first is resumed whenever queue space becomes available and the scheduling policy is re-evaluated at each task spawning point.

3.1 Task Queue Implementation

Achieving scalability of work-stealing implementations in distributed memory environments requires a careful data structure design to minimize locking on the critical path and fast task creation and synchronization primitives. Dinan et al [4] discuss the implementation details of work-stealing in large scale systems and our implementation builds directly on their work.

Figure 3 shows the block diagram of a global task queue distributed on the participating UPC threads. In our implementation, a "global" task queue is stitched from per-thread task queues, managed locally to reduce contention. In the global view, contention overhead is further reduced by splitting the per-thread queue into a private region and a public region [3, 17]; tasks are stolen from the public region and accesses to it are serialized through a lock. Accesses to the private portion of the task queue do not require locking. Moving tasks between the public and the private region is accomplished by updating the split pointer that marks the boundary between these two regions. If space is available, tasks are inserted into the private queue region. When this region is full, tasks are made available for stealing by moving them into the public queue region.

The private portion of the task queue is manipulated like a stack in a LIFO fashion, as executing the most recently created task has a higher chance of exploiting cache locality. The public portion of the queue is manipulated using a FIFO policy, as the oldest task in the queue has the potential to contain the largest amount of work in the task graph. Shared memory runtimes such as Cilk or SLAW steal one task a time, while distributed memory implementations [4] advocate stealing half the number of available tasks. As discussed later, in our implementation we use hardware topology information to provide an adaptive strategy to control the chunk size.

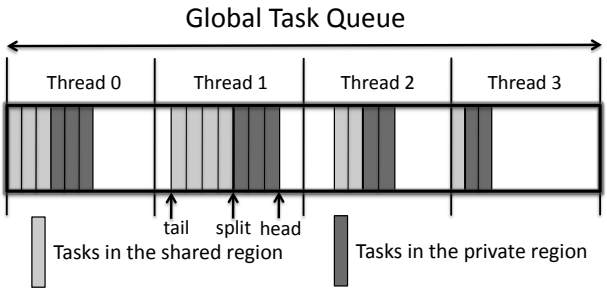


Figure 3. A Global Task Queue Structure in the UPC task library

Besides using an adaptive stealing granularity, we provide extensions to support dependent task execution. The `wait_task_queue` data structure stores the tasks that are spawned, but cannot be scheduled because they are dependent on tasks not yet completed. The `sync_node_map` is a map whose input key is a task and the value is a `sync_node` entry that establishes a relationship between the input key task and the list of tasks that are dependent on the input key task. The `sync_node` entry has a reference counter and a list of dependent tasks stored in the `wait_task_queue`. When a task with an output dependency completes, the runtime library accesses the `sync_node_map` to find the corresponding `sync_node` entry and decrements its reference counter by one. If the reference counter reaches zero, then the dependent tasks of that `sync_node` are ready to be executed. Hence, those tasks in the `wait_task_queue` are moved to the ready task queue so that they can be scheduled for execution.

3.2 Hierarchical Work Stealing

Most of the existing research on work-stealing for shared-memory systems ignore locality concerns or the non-uniform access, hierarchical nature of current architectures. SLAW extends work stealing implementations for shared memory with a notion of locality. In their approach, locality is under programmer control and it is expressed at the application level using the *place* construct. Work-stealing occurs only within a place.

Places provide for a two-level abstract view of the system, local or non-local, while architecturally locality has a more hierarchical nature: shared cache, shared NUMA domain, shared node and inter-node. Yan et al [23] introduce hierarchical place trees abstractions but do not implement the runtime mechanisms required for proper support of work stealing. Our work provides these mechanisms together with a thorough performance evaluation of hierarchical work stealing.

Allowing programmers to specify locality domains provides useful optimization information but can lead to non-portable programs where logical locality domains do not naturally map to hardware. HotSLAW aims to provide support for specifying and composing hierarchical work-stealing schedulers. As the first prototype, we provide a hardware topology aware hierarchical scheduler. This hardware aware scheduling is orthogonal and easily composable with application specific notions of locality.

We propose and evaluate mechanisms for victim selection (task queue) and heuristics to guide the granularity of steal events: the *Hierarchical Victim Selection (HVS)* policy determines from which thread a thief thread steals work and the *Hierarchical Chunk Selection (HCS)* policy dictates how much work a thief thread steals from the victim thread.

Benchmark	Tasks Created	Avg. Task Time	Input / Output Size	Task Creation Ovhd.	Steals	Tasks Serialized (%)
Fibonacci	2,692,537	1.163 μs	4 / 8 bytes	0.172 μs	95	258,928 (8.7%)
NQueens	306,719	23.270 μs	80 / 4 bytes	0.174 μs	47	129,012 (29.6%)
UTS (T1L)	102,181,082	0.089 μs	32 / 0 bytes	0.162 μs	485	93,553,030 (47.8%)
UTS (T2L)	96,793,510	0.114 μs	32 / 0 bytes	0.161 μs	378	82,249,556 (45.9%)
UTS (T3L)	111,345,631	0.075 μs	32 / 0 bytes	0.159 μs	46703	108,983,482 (49.4%)
SparseLU	1,430,912	6.281 μs	16,16,24 / 0 bytes	0.166 μs	2320	1,344,733 (48.4%)

Table 1. Execution statistics on 8 core Nehalem. "Tasks Created", "Steals", and "Tasks Serialized" are the total numbers contributed by all 8 UPC threads. "Input / Output Size" indicates the size of the input and output of the task function in each benchmark (SparseLU has three different task functions). "Tasks Serialized (%)" shows the number of cutoff serialized tasks due to the bounded task queue mechanism and its percentage relative to the total number of tasks.

3.2.1 Hierarchical Victim Selection

Random selection has been the state-of-the-art method in selecting victims for work-stealing in shared-memory architectures. SLAW exploits locality by stealing only from threads within a place, defined as sharing an L2 cache in their study. Of course the size of a place can be arbitrarily extended, but it has the inherent limitation of allowing only descriptions of two level hierarchies.

In HotSLAW, we allow the specification of an arbitrary locality hierarchy directly at the scheduler level. The implementation provides a default hierarchy that mimics hardware locality: cache, socket, node. In the *Hierarchical Victim-Selection (HVS)* policy, a thread first attempts to steal work from the nearest neighbors, and gradually moves up the locality hierarchy. Depending on the number of trial steals at each level of the hierarchy, the policy can be tuned to favor different levels of locality. The default HotSLAW policy for any shared memory (inner) domains is to perform a number of random stealing attempts equal to the number of cores within the domain. When reaching the topmost domain, the implementation performs a tunable number of attempts at that level, before reverting to stealing from the lowest domain. In our implementation, the number of attempts at the topmost level is chosen as $4*log(N)$, where N is the number of cores in the system.. For all benchmarks presented we have obtained good performance results with these default settings. All these settings are also user configurable.

3.2.2 Hierarchical Chunk Selection

Previous work has shown that the performance of work stealing algorithms is sensitive to the number of tasks stolen at each step: this amount is referred to as chunk size.

Work stealing algorithms for recursive parallelism on shared-memory architectures, such as Cilk's runtime system, steal one task from the tail of the victim's queue, hoping to maximize the probability of stealing the task with the maximum amount of work [2]. Particularly, for structured task execution trees where all the leaf nodes have similar depth or all the parent nodes have similar number of child nodes, stealing this single task at the tail of the task queue often makes both the thief and the victim's task queue have similar amount of work.

In order to provide scalability on distributed memory, work-stealing schedulers [4, 17] use a *StealHalf* policy, where thieves steal one half of the victim's queue. The *StealHalf* strategy has been shown to be effective when inter-node work stealing is frequent or when the task execution tree is unstructured and exhibits bursty behavior.

Our approach, the *Hierarchical Chunk Selection (HCS)*, allows for tuning the chunk size for the various levels of the locality hierarchy. Based on the distance between the thief and the victim thread, *HCS* steals a fixed-sized chunk for inner hierarchy levels and uses *StealHalf* at the topmost level, e.g. inter-node. For shared-memory domains, small chunk sizes (1 or 2) have been shown to provide good performance for both structured and bursty/unstructured task graphs. Inter-node work stealing is expensive in a distributed memory environment and *StealHalf* reduces the number of steals.

4. Evaluation Setup

We evaluated the performance of HotSLAW on both shared- and distributed-memory systems. The shared-memory machine is a two-socket quad-core Intel Xeon 5530 (Nehalem) 2.4 GHz system. The distributed-memory machine is a IBM iDataPlex system, each having two quad-core Intel Xeon 5500 (Nehalem) 2.67 GHz processors, for a total of 8 cores per node, connected by 4X QDR InfiniBand technology.

We use four benchmarks: Fibonacci, NQueens, Unbalanced Tree Search (UTS) and SparseLU. Fibonacci recursively creates a number sequence where each subsequent number is the sum of the previous two. NQueens calculates the number of possible solutions to place N queens on a N×N chess board. UTS [16] is a synthetic benchmark specifically designed to evaluate scheduling, load balancing, termination detection and task coarsening strategies. It performs a parallel traversal of an irregular and unpredictable search space. UTS counts the number of nodes in an implicitly defined tree; any sub-tree in the tree can be generated completely from the description of its parent. The number of children of a node is a function of the node's description, which is obtained from its parent node. Load balancing of UTS is particularly challenging since the distribution of sub-tree size exhibits extreme variation; frequent small sub-trees and occasionally enormous sub-trees. The SparseLU (Sparse LU Decomposition) kernel computes an LU matrix factorization. The matrix is organized in blocks and not all blocks are allocated due to the sparsity of the matrix. This sparsity causes load imbalance during computation.

We have developed UPC implementations for all these benchmarks using our dynamic tasking library. On shared memory, we compare the performance of HotSLAW with the performance of OpenMP tasking features: we use implementations from the BOTS (Barcelona OpenMP Task Suite) suite [6]. The OpenMP version of UTS is from the UTS-1.1 distribution [16], which uses its own work-stealing library written in OpenMP. For the evaluation we use the Berkeley UPC compiler and use the gcc version 4.4.3 compiler to build the UPC runtime. OpenMP task applications are compiled with gcc version 4.4.3 and icc version 11.1.

5. Overhead of Work Stealing

Work-stealing runtimes introduce overhead associated with generating and manipulating tasks, as well as overhead due to degradation of task performance when locality assumptions are breached after a steal. Loss of task performance needs to be carefully considered in the UPC environment, where tasks are allowed to carry and perform remote data accesses.

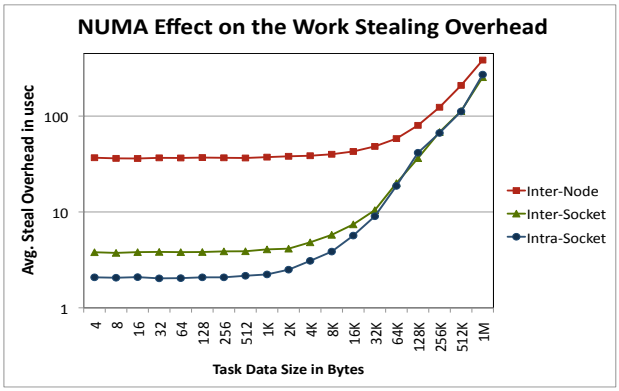


Figure 4. *Average time to steal an empty task with varying input argument sizes on the IBM iDataPlex cluster.*

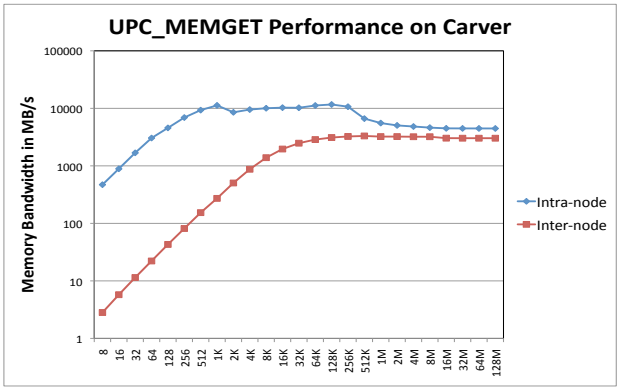


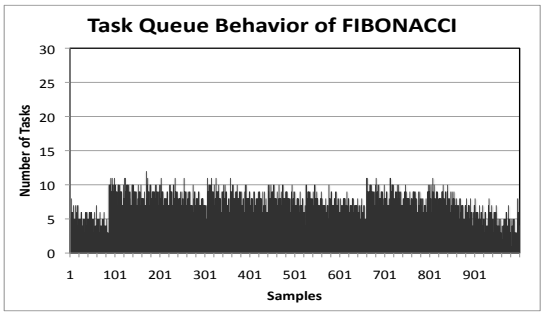
Figure 5. *Memory bandwidth on the IBM iDataPlex cluster. Intra-node measures the inter-socket bandwidth while inter-node measures the InfiniBand bandwidth.*

In Table 1 we present statistics obtained from executing applications on a eight core Nehalem system. As illustrated, creating a task with an eight byte argument payload takes around $0.2\mu s$. To capture the overhead of queue manipulation, we measure the overhead of stealing an empty task on the iDataPlex system. In the benchmark, all tasks but one thief generate work. We vary the size of the input data buffer and present performance numbers for three levels of hierarchy in Figure 4. The overhead of stealing an empty task within the socket is around $2\mu s$, the overhead of stealing inter-socket is around $3.8\mu s$, while the inter-node overhead is $36\mu s$. For empty tasks, lock acquisition dominates the overhead of work stealing. Increasing the task payload to more than 32K bytes shows the effects of bandwidth on the performance of our work stealing implementation.

The *inter-node* work-stealing is 10 to 15 times slower than the *intra-node* work-stealing until the data size reaches 4Kbytes. This gap reduces as the input data size increases such that the *inter-node* is asymptotically just two times slower than the *intra-node*.

Overall, Figure 4 indicates that even when differences in stealing latency within a shared memory hierarchy are observable, their effect is unlikely to cause large end-to-end application performance degradation unless an application has very large number of steals.

UPC applications are often optimized to exploit data locality such that shared data accesses tend to access memory with local affinity. Therefore, stealing a task from a remote node will increase the chance to access the victim's memory. In Figure 5, we show the bandwidth of a *upc_memget* operation obtained within a node and



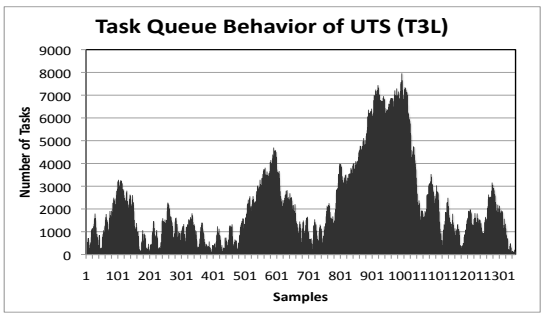


Figure 6. *The number of tasks in the task queue are sampled at every 1000* `taskq_put` *operations.*

across nodes. We see two orders of magnitude difference between *intra-node* and *inter-node* bandwidth when transferring data less than 1KB. This bandwidth gap reduces as the data size increases and, after 1M bytes, *intra-node* bandwidth is consistently 50 % higher than *inter-node*.

Figure 5 suggests that tasks containing fine grained accesses are likely to observe a higher relative impact than tasks performing large transfers.

5.1 Implementing Space Bounds

Work-first and help-first scheduling policies have different performance characteristics and impose different stack and memory bounds. Work-first is optimal for scenarios where stealing is rare and its implementation might overflow the program stack. Help-first is good for scenarios where stealing is frequent.

Figure 6 shows the number of tasks queued in the system during runs of the Fibonacci and the UTS benchmarks. For this experiment we provide unbounded task queues, use help-first and sample a random task queue every 1000 `taskq_put` operations.

These two benchmarks illustrate the two ends of the task queue behavior spectrum. Fibonacci produces and consumes tasks at similar rates, such that less than 10 tasks are maintained in the queue most of the time. On the other hand, UTS with the T3L input shows a bursty task generation pattern with states containing few tasks while others contain thousands of tasks.

Most if not all work stealing implementations use a static memory allocation for task queue management. Besides simplicity of implementation and guaranteeing memory bounds, this approach fits well with practical optimization goals: generating enough work and parallelism at application startup using help-first, then switching to work-first and executing tasks inline to avoid creation and manipulation overhead.

Tree-depth cutoff techniques [12] use this approach to reduce queuing overhead. Programmers often bound manually the depth of the recursion tree, use help-first until reaching a threshold depth then switch to code that amounts to work-first. Lines five through

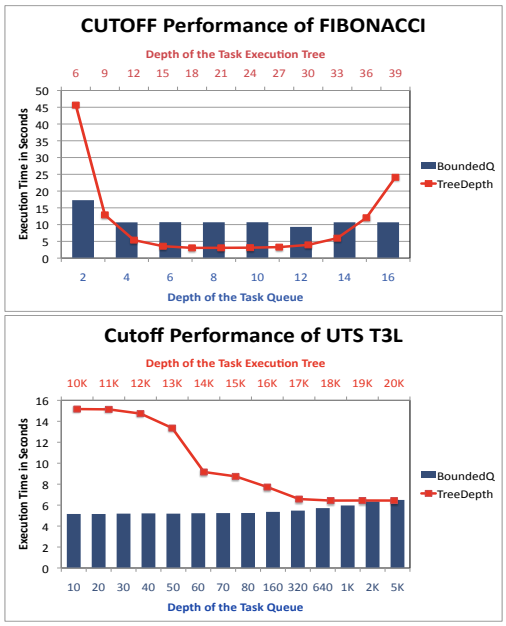


Figure 7. *Performance comparison between the runtime bounded-queue vs. tree-depth cutoff optimization in Fibonacci(47) and UTS (T3L) benchmark*

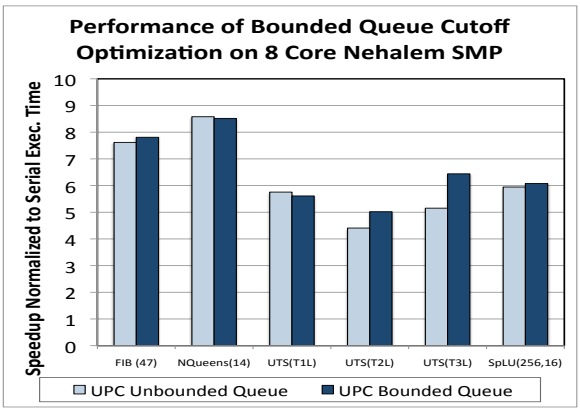


Figure 8. *Performance comparison between HotSLAW with unbounded (UPC Unbounded Queue) and bounded queues (UPC Bounded Queue). Both UPC versions are optimized with tree-depth cutoff optimization. Experiments are run on a 8 core Nehalem SMP.*

eight in Figure 2 illustrate this tree-depth optimization approach. Tree-depth cutoff alleviates task queue memory pressure, but it is oblivious to the internal task queue memory consumption and management. Thus, work stealing environments provide an additional runtime cutoff by imposing task queue bounds and transparently switch scheduling policies whenever queue space is exhausted. This is the approach taken in SLAW and implicitly in HotSLAW.

Figure 7 illustrates the differences between these two approaches. For all benchmarks we implement application level cutoff based on the depth of the recursion tree and run experiments with unbounded task queues. The series labeled "TreeDepth" plots the execution time corresponding to the cutoff setting "Depth of Task Tree". The series labeled "BoundedQ" plots execution time when queues are bound to contain "Depth of Task Queue" entries. For this setting the tree-depth cutoff is disabled.

For Fibonacci, we observe a 15X performance difference between the best and worst tree-depth cutoff (TreeDepth) setting. For runtime bounded-queue (BoundedQ) technique, we observe at most 2X performance difference between the best and the worst setting. For UTS, we observe 2X performance differences for TreeDepth and at most 20% for BoundedQ. For Fibonacci, TreeDepth provides best performance, while for UTS best performance is provided by BoundedQ.

In Fibonacci, the best tree-depth cutoff performance at the tree depth of 21 is three times faster than the best runtime bounded-queue cutoff performance at the task queue depth of 12. Fibonacci has a structured task execution graph whose parent nodes have the same number of child nodes. With tree-depth cutoff, tasks are executed inline directly in the application, while with runtime bounded-queue cutoff, they are executed inline inside the `taskq_put` library call. The overhead of the additional function calls explains this difference.

Contrary to Fibonacci, the UTS benchmark shows that the best bounded-queue cutoff at the task queue depth of 20 is 24.9% faster than the best tree-depth cutoff at the threshold depth of 22,000. This is because the UTS benchmark with the T3L input has a very deep and unstructured task graph and the tree-depth cutoff prematurely serializes a large sub-tree and hence causes a load imbalance. On the other hand, the bounded-queue cutoff method does not serialize a whole sub-tree and instead it serializes one task at a time.

We have repeated these experiments for the NQueens and SparseLU benchmarks. Overall, tree-depth cutoff produces large performance variations and the per application optimal threshold (usually recursion depth) has a wide range. On the other hand, bounding the task queue size provides good performance and little variation within a small range of values for optimal queue sizes. Overall, our results indicate that setting a queue threshold of 40 active tasks provides good results in practice for the workload considered. The types of parallelism present in the benchmarks has been discussed in this section. For reference, the average task length on a eight core Nehalem is shown in Table 1.

Figure 8 shows the performance impact of bounding runtime queues to 40 slots per thread. For each benchmark implementation, we perform a search to detect the best tree-depth cutoff threshold and use this setting for comparisons. This process is commonly used in practice by application developers since the tree-depth cutoff threshold can greatly affect performance of recursive parallel applications.

The series labeled "UPC Bounded Queue" shows the effects of enforcing queue bounds on the performance of the UPC implementation. Note that the SparseLU benchmark has parallel-for parallelism instead of recursive fork-join parallelism. Hence, it is not optimized with tree-depth cutoff.

Imposing queue bounds in HotSLAW does not affect the performance of the FIB, NQueens and UTS(T1L) benchmarks. These have well "balanced" task graphs and the tree-depth cutoff is able to throttle task generation at the right moment. For UTS(T2L, T3L) which have deep unstructured task graphs, bounding the queues provides additional performance improvements of up to 18%. For SparseLU, the benchmark generates a small number of coarse-grained tasks and the bounded-queue cutoff improves performance only slightly.

5.2 Impact of Hierarchical Victim Selection

Figure 9 shows the impact of hierarchical victim selection (HVS) on the performance of HotSLAW on shared memory architectures. As a frame of reference, we present the performance of hand tuned OpenMP implementations from the BOTS [6] suite or developed by independently for UTS [16]. These implementations use tree-depth based cutoff and we select for comparison the best performance obtained after searching the cutoff parameter space. For lack of space,

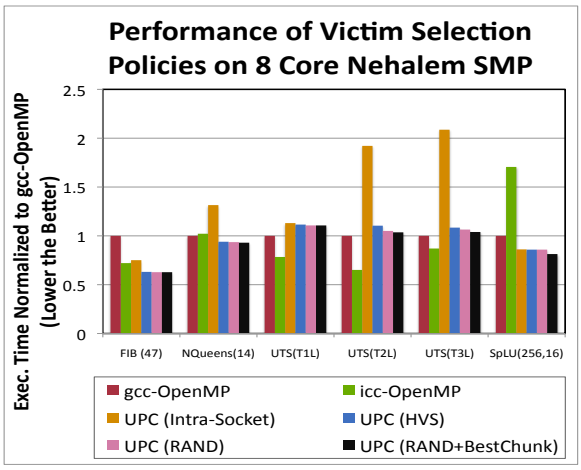


Figure 9. *Performance of OpenMP and victim selection strategies on a 8 core Nehalem SMP. Execution times are normalized to the execution time of the gcc-OpenMP version (lower the better). All UPC versions use a fixed chunk size of 1, except the UPC (RANDOM+BestChunk) uses the best chunk size searched. For reference, the raw speedup numbers of UPC (HVS) scheme for (FIB, NQueens, UTS(T1L), UTS(T2L), UTS(T3L), SparseLU) are (7.7, 7.9, 7.5, 7.2, 7.5, 5.9), respectively.*

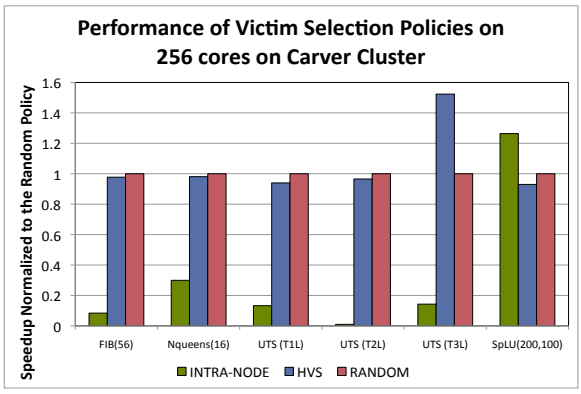


Figure 10. *Performance comparison of stealing strategies on distributed memory: 256 cores on the Carver IBM iDataPlex cluster. The speedup numbers are normalized to the speedup of the random policy.*

we do not include details about the OpenMP implementations, note that it has been shown that performance varies manifold [5, 18] depending on the cut-off parameter selection. For all UPC implementations we present performance when stealing one task per event (chunk size of 1), which is the default value in shared-memory work-stealing runtimes.

The series labeled "Intra-Socket" presents the performance when HotSLAW is configured with a single level hierarchy, contained with a NUMA domain. The series labeled "HVS" presents the performance when using a two level hierarchy spanning the whole node. For reference, we show the performance of HotSLAW when using a randomized stealing policy (used as default in many systems) with different chunk-size granularities: we show data for the default value of 1, as well as for the best value determined through searching.

As our results indicate, HotSLAW provides comparable or better performance than the OpenMP implementations. HotSLAW outperforms both `gcc` and `icc` OpenMP versions in FIB, NQueens, and SparseLU. In UTS benchmarks, `icc` OpenMP shows the best performance and HotSLAW and `gcc` OpenMP have similar performance. For the benchmarks studied, confining stealing only within one socket reduces performance when compared to whole node stealing. When allowing node wide stealing, refining the hierarchy levels does not change performance and HotSLAW with HVS achieves the same performance as HotSLAW with random stealing. This behavior is caused by the fact that most benchmarks use irregular data structures with no data locality. Similar results were obtained by Guo et al [10] for SLAW where they report improvements only when using places for dense algebra codes that fit in cache.

Figure 10 shows the performance of HotSLAW on the IBM iDataPlex cluster, using 256 cores. Again, we perform a search for selecting the setting for chunk size that provides the best performance and use this value for comparisons. The series "Intra-Node" uses a setting where work is hierarchically stolen only within a shared memory node, the series "HVS" uses hierarchical victim selection, and the series "Random" uses the default random stealing policy. For each setting, we present speedup normalized to the speedup of the random victim selection policy.

The first two benchmarks, FIB and NQueens, which have infrequent steals with small input arguments and no shared data accesses within a task function, show that both HVS and Random perform equally, while Intra-Node works poorly. The UTS (T3L) benchmark has frequent steals, which makes the proposed HVS significantly outperform both Intra-Node and Random policies. In particular, HVS is able to provide a maximum 52% speedup when compared to Random. The SparseLU benchmark exhibits a relatively large number of steals and performs multiple shared data accesses within its task functions. In this case, containing the locality hierarchy within a node provides 20% better performance than the other settings. HotSLAW with HVS is able to match the performance of Intra-Node for SparseLU when tuning the number of steal trials within a shared memory level.

For reference, the raw speedup numbers of HVS schemes on 256 cores for (FIB, NQueens, UTS(T1L), UTS(T2L), UTS(T3L), SparseLU) are (179.5, 247.8, 200.1, 222.5, 98.8, 32.1), respectively.

These results indicate that in HotSLAW we can handle well both recursive fork-join parallelism and parallel-for, using only small queue bounds and independent of the inherent load balance present in the application. Furthermore, imposing queue bounds eliminates the need for manual application tuning: HotSLAW is always able to improve performance independent of the tree-depth based cutoff. In particular, due to the tight space bounds, HotSLAW provides similar performance independent of the application level tuning.

5.3 Impact of Hierarchical Chunk Selection

Figure 11 shows the impact on application performance of stealing with different granularities (ChunkSize). In this experiment we use HVS with a single fixed chunk size for all levels of the hierarchy. We report performance on 256 cores of the IBM iDataPlex cluster.

FIB shows the best performance at "ChunkSize = 1" and its performance drastically drops for larger values. FIB has a structured task graph with a well balanced computation where stealing more than one task causes actual load imbalance. The performance of NQueens, UTS(T1L), UTS(T2L) and SparseLU is relatively insensitive to the variation of ChunkSize. Note that performance degradation sometimes occurs for ChunkSize=16: considering the default queue bounds of 40 (public queue size of 20), this setting implements a policy even more aggressive than StealHalf [4]. UTS(T3L) exhibits different trends than all other benchmarks: in this case "ChunkSize=16" provides best performance. For this particular T3L input, UTS generates an unstructured task graph with

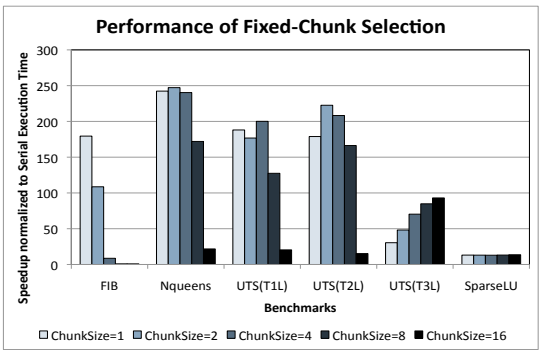


Figure 11. *Performance of fixed chunk selection (Fixed-Chunk) method with varying chunk sizes on 256-core Carver IBM iDataPlex cluster*

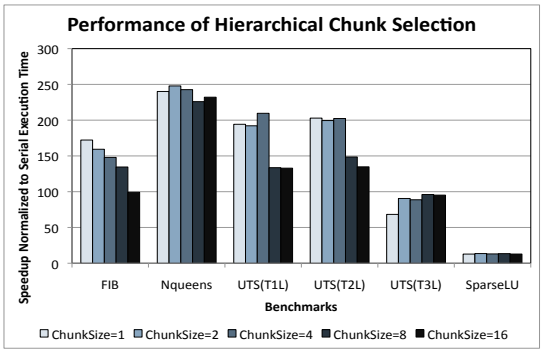


Figure 12. *Performance of hierarchical chunk selection (HCS) method with varying chunk sizes on 256-core Carver IBM iDataPlex cluster*

bursty sub-tree patterns, which makes larger chunk sizes more efficient for consuming the excess number of tasks in the queues.

In Figure 12, we present the performance of the *hierarchical chunk selection (HCS)* method. In this case, we present results for a setting using a single granularity (fixed chunk) for all intra-node steals and *StealHalf* for any inter-node steals. As shown, the overall performance trends are similar to the pure fixed chunk method and *HCS* is able to lower the performance impact when poorly selecting the steal granularity. For example, in the *Fixed-Chunk* mode, the performance of FIB, NQueens, UTS(T1L), and UTS(T2L) drops drastically for ChunkSize ≥ 8, while *HCS* maintains a reasonable level across all chunk sizes.

Figure 13 summarizes the performance trends observed for all possible chunk selection methods. "Fixed-Chunk (Best)" denotes the best performance obtained when using *fixed chunk* and searching through all the possible settings. *StealHalf* denotes the performance when using *hierarchical victim selection (HVS)* with a steal-half policy at each hierarchy level. For applications with structured task graphs, such as FIB, NQueens, and UTS(T1L), both *Fixed-Chunk* and *HCS* strategies outperform *StealHalf*. *StealHalf* is better at frequent stealing operations and unstructured and bursty task patterns, such as UTS(T3L). *HCS* shows the performance when using a fixed chunk size of 1 for its intra-node stealing and the steal-half policy for its inter-node stealing for all the benchmarks. For reference, we compare performance with the random victim selection with *StealHalf* chunk policy setting used by Dinan et al, denoted as "Random+StealHalf".

The geometric mean speedups of *Fixed-Chunk (Best)*, *StealHalf*, *HCS* and *Random+StealHalf* is 116.4, 94.9, 106.1, and 83.3, respectively. *Fixed-Chunk* shows the best performance attainable

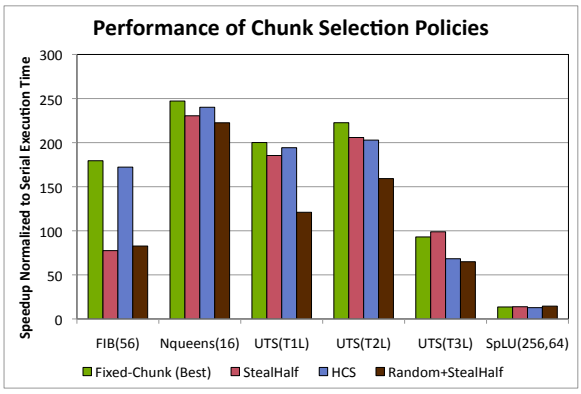


Figure 13. *Performance of different chunk size selection policies on 256 cores Carver. Fixed-Chunk (Best) uses the best chunk size for each application and HCS uses a chunk of size one for its intra-node stealing. Fixed-Chunk (Best), StealHalf, and HCS use HVS for its victim selection, while Random+StealHalf uses random victim selection.*

when manually searching the space of possible setting of the ChunkSize parameter. For reference, the raw speedup of *HCS* and *Random+StealHalf* on 256 cores for (FIB, NQueens, UTS(T1L), UTS(T2L), UTS(T3L), SparseLU) are (172.2, 240.1, 194.2, 202.9, 68.3, 12.8) and (82.7, 222.5, 121.0, 159.2, 64.9, 14.5), respectively. On average, *HCS* performs 27% better than *Random+StealHalf* method.

While the *Fixed-Chunk* method can attain the highest peak performance, it is also sensitive to the value chosen for ChunkSize and it can perform very poorly for the wrong choice. The *StealHalf* method does not require any tuning as it always steals half of the victim's tasks. However, its peak performance is lower than *Fixed-Chunk* and *HCS* for applications with structured task trees because the *hierarchical victim selection (HVS)* optimization, which we described in Section 3.2.1, forces *StealHalf* method to steal from the intra-node most of the time. The peak performance of the *HCS* method is close to the peak performance of the *Fixed-Chunk* method because of the *HVS* optimization. On the other hand, among these three methods, the *HCS* method shows the highest average performance for all chunk sizes, which shows the robustness of the proposed technique.

6. Related Work

Dynamic tasking and work stealing runtimes have been extensively studied. Recent work presents scalable and locality aware implementations for both shared- and distributed memory systems. SLAW (Scalable Locality-Aware Work-stealing) [10] is designed to support the Habanero-Java [1] task-parallel language. Habanero-Java supports locality control using places and SLAW performs work stealing only within a place. Guo et al. show performance improvements when enabling cache locality for dense linear algebra kernels. Similar to SLAW, our approach uses an adaptive work stealing policy. In addition, HotSLAW provides a locality hierarchy with an adaptive stealing granularity policy.

The Scioto framework [3] provides locality-aware dynamic load balancing and inter-operates with MPI [7], ARMCI [14], and Global Arrays [15]. Scioto implements locality-awareness by prioritizing the task queue; tasks with high local affinity are placed toward the head of the task queue and tasks with low local affinity are placed toward the tail of the task queue. High local affinity tasks are executed by the local thread while the low local affinity tasks are stolen by other threads. Affinity in Scioto is declared with respect to a particular process and the mapping of processes

to cores is left unspecified. In contrast, our default locality notion takes into account the hardware memory hierarchy. Dinan et al. [4] present a scalable implementation of work stealing for ARMCI. Our implementation is directly based on theirs and it extends it with hierarchical victim and chunk selection.

Very recently, Saraswat et al. [20] introduce lifeline based load balancing that proposes an alternate approach to random stealing on large scale systems. Their main goal is to minimize the impact of missed steals on performance and their work introduces the notion of lifeline graphs. A lifeline graph provides a meta-topology used for work stealing: when a node fails to steal, it quiesces and informs the outgoing edges in the lifeline graph. Work arrives from a lifeline and it is pushed by nodes onto all their active outgoing lifelines. To implement lifeline graphs the authors advocate cyclic hypercubes: it is not clear how well these graphs map onto hardware locality domains.

The PGAS programming model presents the programmer with a single global address space, with efficient one-sided access to global data. The one-sided data access model provides a natural way to implement [17, 21] work stealing operations. Shet et. al. [21] proposed the Asynchronous Partitioned Global Address Space model (APGAS) and Asynchronous Remote Methods as potential extensions to the UPC language standard. They implement a fixed-size global task queue with an estimated upper bound for APGAS support.

Olivier and Prins [17] developed an optimized application level runtime library for the UTS benchmark. The library is written in UPC and they discuss multiple implementation aspects. In particular, their approach provides the first discussion of the importance of quiescing threads that fail to steal and provides the seed for lifeline based load balancing. Olivier and Prins describe an implementation where tasks mark their intention to steal and work is pushed from producers. Our implementation uses a pull based approach with locking. We plan to extend our implementation with a similar approach and provide a performance comparison. Their distributed-memory implementation uses the half-chunk policy for work stealing.

Scheduling for shared memory work addressed the problem of dynamically scheduling for-all loops with unpredictable iteration execution times; Guided self-scheduling [19] and Taper [13] choose large chunk sizes at first to reduce the overhead of scheduling, small chunk sizes later for load balance. These loop scheduling approaches assume that the number of tasks is a priori known and the task pool only shrinks after initialization.

A recursion tree-depth based cut-off mechanism is an important overhead reduction technique, especially for fine-grained tasks. Besides the techniques described for SLAW, there exists a large body of work applied to OpenMP implementations. Duran et. al [5] proposed an adaptive cut-off technique for OpenMP implemented in the Nanos OpenMP environment. Their cut-off mechanism uses runtime profiling to decide whether to create or serialize a new task. Their profiling technique assumes that all tasks at a given recursion level will have a similar behavior, a condition that will not hold for unstructured and bursty task graphs.

Zheng et al. [24] developed a hierarchical load balancing for Charm++ [11] applications using an object based data model. They divide the processors into hierarchical autonomous groups, thereby decentralizing the load balancing task. Charm++ uses runtime monitoring to periodically assess the computational load for each object and builds a load database. The runtime consults this load database to determine load imbalance and migrates computation objects to low-loaded processors.

7. Conclusion

While PGAS languages provide support for irregular data structures, the SPMD instances of these languages do not support irregular computational structures. This is increasingly important as people parallelize a wider class of applications, and there are growing concerns about performance heterogeneity of large-scale systems. In this paper, we presented HotSLAW, a dynamic tasking library for the Unified Parallel C programming language. HotSLAW provides a simple and effective way of adding task parallelism to SPMD programs that takes advantage of the global address space both in the implementation and expression of tasks, which may contain global pointers that are location independent. To exploit locality in distributed-memory many-core clusters, we presented two hierarchical work-stealing optimization techniques: *hierarchical victim selection (HVS)* steals work from the nearest available victims to preserve locality and *hierarchical chunk selection (HCS)* dynamically determines the amount of work to steal based on the locality of the victim thread. HotSLAW allows UPC programmers to selectively load balance all or parts of their computation, and it is designed to work with multiple UPC implementations. We evaluated HotSLAW performance on both shared and distributed memory architectures. On shared memory systems, HotSLAW provides performance comparable or better than manually optimized OpenMP implementations, improving by as much as 109%. On distributed-memory systems, *HVS* is able to improve performance by up to 52% when compared to the default random victim selection and *HCS* provides performance improvements up to 122% compared to the *StealHalf* method. For the workload considered in this paper, the combination of *HVS* and *HCS* enables HotSLAW to achieve 27% better performance than the state of the art approach using a *random* victim selection and a *StealHalf* strategy

References

[1] "The Habanero Java (HJ) Programming Language." [online] Available:. http://habanero.rice.edu/hj.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[3] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A Frameworkfor Global-View Task Parallelism. In *Proceedings of the 37th Intl. Conference on Parallel Processing (ICPP)*, 2008.

[4] 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, pages 1–11, New York, NY, USA, 2009. ACM.

[5] A. Duran, J. Corbalán, and E. Ayguadé. An Adaptive Cut-off for Task Parallelism. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC'08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[6] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page 124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.

[7] T. M. Forum. MPI: A Message Passing Interface standard, 1993.

[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, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09*, pages 1–12, 2009.

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

[11] 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*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[12] H.-W. Loidl and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Proceedings of the Glasgow Workshop on Functional Programming*, pages 8–10, Ullapool, Scotland, July 1995.

[13] S. Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 200–211, New York, NY, USA, 1992. ACM.

[14] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. In *IPDPS'99*, pages 533–546, 1999.

[15] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: a portable "shared-memory" programming model for distributed memory computers. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 340–349, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[16] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 123–131, Washington, DC, USA, 2008. IEEE Computer Society.

[18] S. Olivier and J. Prins. Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs. In *IWOMP'09*, pages 63–78, 2009.

[19] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36:1425–1439, December 1987.

[20] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[21] A. G. Shet and R. J. Harrison. Asynchronous Programming in UPC: A Case Study and Potential for Improvement. In *The 1st Workshop on Asynchrony in the PGAS Programming Model (APGAS)*, 2009.

[22] UPC Language Specification, Version 1.0. Available at http://upc.gwu.edu.

[23] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing*, pages 172–187, 2009.

[24] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 436–444, Washington, DC, USA, 2010. IEEE Computer Society.