Scalable Lightweight Task Management for MIMD Processors

Daniel G. Waddington, Chen Tian
Computer Science Lab
Samsung R&D Center, San Jose, CA
{d.waddington, tianc}@samsung.com

KC Sivaramakrishnan
Computer Science Department
Purdue University
chandras@cs.purdue.edu

ABSTRACT

Multiple Instruction Multiple Data (MIMD) processors are becoming increasingly viable for realizing enhanced performance in embedded applications, particularly for compute intensive applications that do not lend themselves to GPU-like (SIMD) processors. Like their SIMD counterparts, MIMD architectures demand careful consideration of software architecture and design. In this paper, we focus on lightweight task management that is typical of many parallel programming solutions such as OpenMP, Cilk, Intel TBB.

As a representative MIMD embedded processor, we compare the performance and scalability of multiple designs within the context of a 64-core TILE processor from Tilera Corp. We present experimental data collected from the measurement of performance for user-level cooperative scheduling that we are developing to support parallel programming extensions to C/C++. We analyze the performance of well-understood scheduling schemes, such as multi-queue work-stealing, as well our own designs that are specifically tailored to the underlying hardware and are also designed to adapt to changing application requirements that often occur at runtime.

1. INTRODUCTION

Multiple Instruction Multiple Data (MIMD) designs form the basis of most non-GPU multicore processors. A typical MIMD processor is made up of multiple CPU cores that are arranged on-chip and interconnected using a high-speed switched network. It may or may not be fully cache coherent. The interconnect serves both core-to-core communications as well as data transfers between memory controllers and I/O systems. This hardware design is aimed at sustainable performance as the number of cores scales to tens, hundreds and even thousands.

This type of architecture requires careful consideration of software design and implementation with respect to parallel execution. The conventional approach to building concurrent software is through the use of multithreaded programming APIs such as those found in POSIX-based operating systems. Nevertheless, multithreaded programming is relatively heavyweight and does not scale well for large numbers of tasks as complexity becomes difficult to manage. One of the key reasons for this is that multithreaded programming APIs provide a very low level of abstraction and put the onus on the developer to ensure that they are used correctly; a task that becomes inherently difficult in the manycore arena.

An alternative approach to concurrent software development is to use language-level solutions that explicitly provide support for concurrency. Example languages and extensions include X10 [6], OpenMP [13], Intel's Cilk and Thread Building Blocks (TBB) [14].

The focus of this paper is the design and evaluation of an adaptive task management scheme that reconfigures the underlying task management queuing scheme in response to changes in the "pressure" of queues detected through runtime monitoring. We present experimental results and a quantitative evaluation of different schemes within the context of the Tilera TILEPro64 MIMD processor.

2. BACKGROUND

2.1 SNAPPLE Language and Runtime

The work discussed in this paper is being carried out within a broader effort to develop language, run-time and operating system capabilities for next-generation MIMD many-core processors.

As part of our solution, we are currently developing a C++-based parallel language, compiler and runtime called SNAPPLE, that provides specific capabilities for real-time processing and QoS (Quality-of-Service) management. SNAPPLE also provides language constructs (augmenting C++) for lightweight task management and safe data sharing. It is implemented using the LLNL ROSE compiler [15] as a source-to-source transformation. SNAPPLE code is compiled to C++, which is in turn compiled to the platform using GNU gcc. Currently SNAPPLE is only supported in cache-coherent SMP environments. We are exploring the extension of SNAPPLE technology to non-coherent environments such as Intel's Single-chip Cloud Computer (SCC). A full description of the SNAPPLE language is outside of the scope of this paper.

Lightweight tasks provide the fundamental means to create concurrency in a SNAPPLE program. The basic construct for lightweight tasks is the *async–finish* block, inspired by the X10 programming language [6]. This construct provides a mechanism for task creation and synchronization,

as well as a scoping mechanism for data sharing/copying. The following excerpt illustrates the use of these constructs in a parallel binary tree search:

```
void search(Tree* node, int v) {
  exit_if_so;
  if (node->val == v) {
    result = node; /* shared type variable */
    terminate;
  }
  finish {
    async { search (node->left, v); }
    async { search (node->right, v); }
  }
}
```

The *async* keyword defines a new lightweight task that can safely execute in parallel with the calling task, as well as with other async-created tasks. The *finish* construct defines an implicit barrier whereby the code that follows the finish block is only executed after all of the tasks spawned within the finish block run to completion. Lightweight tasks are transparently mapped to different system threads by the SNAPPLE runtime. The SNAPPLE compiler builds the necessary code to create lightweight task objects and schedule them accordingly. System threads, that are bound to specific cores, service the tasks.

In the current implementation, new stack space is allocated for each newly created lightweight task. To avoid explosion of memory usage in deeply recursive programs, SNAPPLE uses a dynamic thresholding technique that limits the memory footprint by resorting to serial execution (i.e., no tasks are created) when a given memory threshold is reached.

SNAPPLE also provides primitives for task cancellation. Exit points designated through *exit_if_so*, define a point at which a task checks (polls) for an exit notification, whilst *terminate* notifies all tasks spawned within the same finish block scope to terminate at the next exit point. Exit notifications are also propagated down into nested async-finish blocks.

In order to avoid concurrency bugs such as data races and deadlocks, inter-task data sharing is strictly enforced through the use of *shared memory types*, which provide monitor-like access protection. All access to shared data is implicitly protected. Multiple shared data instances can be manipulated with *atomic sections*, which automatically enforce mutual exclusion by implicitly acquiring the necessary locks before entering the atomic region and releasing on exit. Locks taken by atomic scetions are automatically sorted in order to prevent deadlock.

One of the key design features of SNAPPLE and its runtime is the ability to flexibly change the underlying task management scheme. Specifically, different queue arrangements and scheduling policies can be easily "plugged in" depending on specific application and platform requirements. This capability has allowed us to easily explore design tradeoffs with respect to different task management schemes.

2.2 TILE64 MIMD Processor Architecture

We use TILEPro64 64-core MIMD processor in this work. It is a cost effective platform that delivers high performance as well as power efficiency. This processor is a 64-way MIMD design, arranged in an 8x8 grid (see Figure 1). Each core is 32bit VLIW, operating at 866MHz in around a 0.5W power envelope. The total theoretical maximum throughput is 443 billion operations per second. The TILE processor provides a user-programmable switched interconnect, as well as fully cache coherent shared memory. Four separate memory controllers manage a total of 8GB of off-chip DDR2 memory.

The TILEPro64 processor integrates four 64-bit on-chip DDR2 memory controllers; each controls a separate DRAM device. Virtual-to-physical address space mappings are configured by the hypervisor. In general, the operating system attempts to balance accesses by distributing physical memory allocations across the controllers. The default policy is to allocate from the controller connected to the respective quadrant.

The TILEPro64 is a fully cache coherent architecture. Coherency is maintain via a dedicated interconnect channel that supports communications between each tile's cache engine. Cache lines are "homed" on a specific tile. This is by default the tile that allocated the page of memory. However, the TILEPro64 also supports a feature known as *hash-for-home* that uses a hash function on the physical address to govern home tile selection, as well as memory striping whereby each single page is spread across all four of the memory controllers. Tiles that have copies of cache lines, but are not the respective home tiles, are known as "sharing" tiles.

When a tile T accesses data in a cache line X it first checks its own local L1 and L2 caches. On a cache miss, or if the copy has been invalidated, the tile then makes a request to the home tile, H, for the cache line. Tile H will optionally load the cache line from main memory and then send the complete line back to the requesting tile T. Similarly for writes, the tile T will send write requests to the home tile H and wait for write acknowledgment once other copies of the line in sharing tiles are invalidated.

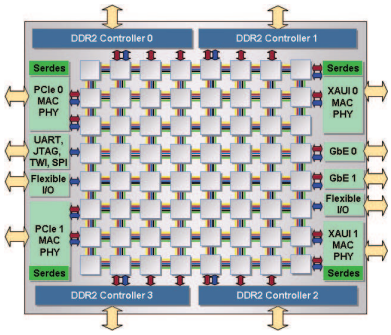


Figure 1: TILEPro64 Architecture

In our prototype architecture, each core runs a single system thread, which services lightweight task contexts that are queued in the system. Each system thread is pinned down to a specific core by setting the thread's affinity mask. Each core is explicitly reserved for the SNAPPLE runtime by configuring the processor's hypervisor accordingly. These reserved cores are not visible to the Linux OS and are not used to service kernel tasks or interrupts.

2.3 Existing Task Management Schemes

There are a number of ways in which task queues can be distributed and/or shared across multiple cores/threads. Common topologies are shown in Figure 2. The most straightforward of these to implement in a shared memory MIMD environment is a single global queue that is shared across multiple threads running on separate cores (refer to Figure

2a). This scheme has the advantage that there is no overhead associated with the migration of tasks. In low-load situations (where load pertains to tasks creation/servicing volume) this approach works well and minimizes the "cost of concurrency". Nevertheless, in heavy load or large numbers of cores, the centralized queue can become a point of contention; intense producer/consumer pressure on a single queue can result in significant performance degradation.

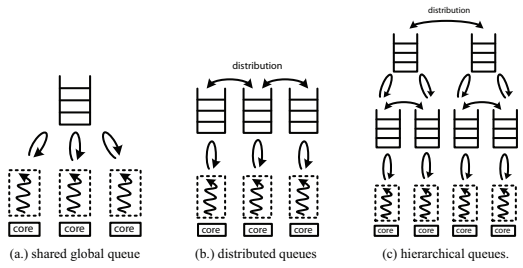


Figure 2: Queuing Topologies

Historically, more attention has been given to distributed queuing schemes whereby each thread in the system has a local queue which it services (refer to Figure 2b). Tasks are load balanced across the queues by task migration either by being pushed by the task creator (work-sharing) or pulled by the task consumer (work-stealing). Distributed queuing schemes help to alleviate *pressure* on the queues in high-load situations by reducing producer/consumer contention. However, overhead is incurred through additional operations required for load balancing and task migration. They do lend themselves to distributed memory systems such as those found in large HPC clusters. It is for this reason that we believe prior work has focused more on distributed queuing schemes. From the context of embedded systems and potentially low-load applications they may not be the best choice for optimal performance.

Queues can also be arranged hierarchically (refer to Figure 2c) so that stealing is spatially localized among groups of queues before coarser-grained stealing across higher-level queues.

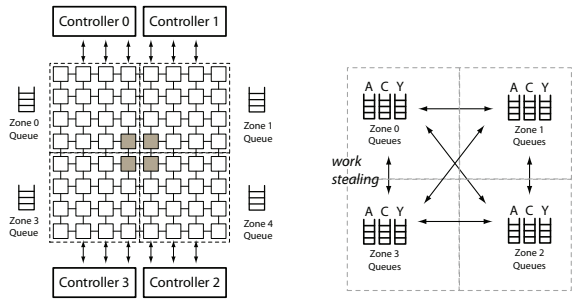
3. TASK MANAGEMENT OPTIMIZATION

This section describes our new task management architectures that are explicitly designed to take advantage of Tilera-like MIMD processor designs.

3.1 Tasks and Queues

The SNAPPLE runtime manages three kinds of tasks; *asynchronous* (A), *continuation* (C), and *yields* (Y). A-type tasks are created for each *async* construct. C-type tasks are created when a thread reaches the closing scope of a *finish* block and detects that nested asynchronous tasks still need to be serviced. Finally, Y-type tasks are created in response to yield operations. We manage these in separate queues (A,C,Y) to help reduce stalling due to dependencies and also to prioritize between yielded and non-yielded tasks. A-queues are serviced at the highest priority. Continuations are not placed on the C-queue until they are ready to execute, that is, all respective asynchronous tasks have completed.

3.2 Zone Queue Architecture



(a) Topology (b) Stealing

Figure 3: Memory Zone Queueing

The zone queue scheme arranges task queues according to memory controller topology and maintains exactly one set of queues for each memory controller and tile quadrant (see Figure 3a). Tasks created by a thread executing on a given tile are placed on the A, C or Y queue for the zone in which the tile resides. In servicing tasks, a tile first checks the A queue belonging to the local-zone. If the queue is empty, then A queues in adjacent zones are checked using a nearest-neighbor first policy. For example, a tile in zone 0 will first check zone 1 and 3, and then finally zone 2 (refer to Figure 3b). If all of the A queues are empty, then the thread will attempt to service C and Y queues respectively. This scheme also requires that memory allocated by a task is taken from the nearest memory controller for the quadrant. If the local memory banks are fully allocated, then an adjacent bank can be used.

The zone-queue architecture brings performance benefits to heavily loaded and data intensive applications (experimental results are given later in Section 4). This approach localizes cache and memory traffic (including coherency traffic) by reducing the need to communicate across the tile array particularly with respect to cache misses passing all the way through to main memory. Furthermore, it helps to improve cache performance by localizing consumer/producers and helping to increase the likelihood of cache hits for task data structures. Task management overhead is also reduced by limiting the average migration distance of tasks between cores.

The concept of zone-queuing can be easily extended to support hierarchical queues that are structured according to core, memory and cache topologies. For example, the scheme in Figure 3 could be extended to support hierarchical queues for 1, 2x2, and 4x4 zone topologies.

3.3 Adaptive Queuing Scheme

As previously discussed, queuing architectures range from a single coarse-grained shared queue, with no need for work-stealing, to fine-grained per-core queues coupled with work-stealing. As discussed previously in Section 2.3, heavy task loads are better suited to schemes that support multiple queues and thus relieve pressure on any single queue. Nevertheless, fine-grained queues come at a cost. Checking of multiple queues and migration of tasks between them all cost overhead. In light loading situations, coarse-grained queues can provide efficient task management scheme with least incurred overhead. If the approach is too fine-grained

however, excessive task migration distance can also impact performance.

The problem of deciding queue-granularity becomes an issue of application behavior with respect to task creation, which of course may potentially change over time. To help address this problem we introduce an *adaptive* queuing scheme that changes the queue arrangement and granularity over time in response to runtime pressure measurement in the system.

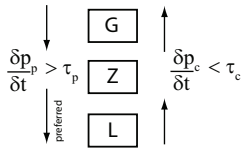


Figure 4: Adaptive Scheme States

Our approach is to periodically sample queue pressure and modify the granularity accordingly. In our current prototype we adapt between three queue schemes: global (G), zone (Z) and local (L). Each quadrant can independently transition between these schemes. Producer pressure, given by

$$P_{producer} = \frac{\delta p_p}{\delta t} \quad (1)$$

is measured by the number of queue locks that incur at least one failure (i.e., the queue lock was held by some other thread). Consumer pressure, given by

$$P_{consumer} = \frac{\delta p_c}{\delta t} \quad (2)$$

is measured by counting the number of task steals performed by a thread.

The pressure counts are maintained in a global array with per-thread entries. An "adaptation controller" thread periodically samples the count and triggers any required change in adaptation scheme. Pressures are compared against predefined threshold values, τ_p and τ_c. In the current prototype implementation these values are static and are tuned by hand for a given application. For the tests we used a threshold value of 16 events per 100 msec sampling period. In future work we envisage the use of auto-tuning like approaches to optimization of threshold values.

The adaptive scheme allows independent arrangements for different zones. For example, tiles in zone 0 may use a global queue scheme while tiles in zone 1 use a zone queue scheme. There is always only one global queue and thus any zone using a G scheme will share the same queue.

4. EXPERIMENTAL RESULTS

This section presents experimental data that quantifies the benefit of the different task management schemes on the TILE processor. The objective of our experimentation is to measure two key aspects of the solution: 1.) The performance of the different schemes for different types of processing, and 2.) The scalability of each scheme with respect to increasing number of cores.

For the experimental results, we used the TILEPro64 processor, with Tilera MDE 3.0.alpha2 which includes an early port of GNU gcc (4.4.3). The system was configured with Zero-Overhead Linux (ZOL) with all but four of the tiles instantiated in data plane mode making them exempt from interrupt processing and other OS processing activities.

Table 1: Benchmark Applications

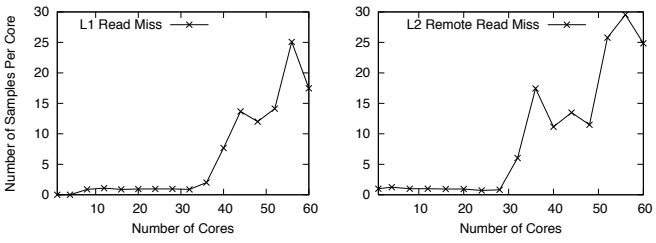
Name	Peak Cores	Version
fib-30	60	Calculation of the 30th Fibonacci number.
fft	48	Fast-Fourier Transform (Cooley-Turkey algorithm).
kronecker	60	Matrix Kronecker product on 2D 32-bit integer matrices; 20K source elements. Memory caching turned off.
sort	36	Parallel sort of randomly (deterministic across runs) generated 10K integer vector. Algorithm based on merge of 100 element sorted sublists.
image	32	Image processing (filtering and segmentation).
indepwork	60	Independent work in the form of PI calculations.

4.1 Benchmark Performance

We developed six benchmarks applications for the evaluation (refer to Table 1). The benchmarks are representative of the types of processing that we would expect to see in our current products. Each benchmark was implemented in SNAPPLE.

For each experiment, we ran the benchmarks with the peak number of cores that demonstrated maximum speed-up across both number of cores and queuing schemes. These values are given in Table 1. We chose to take this approach because some of the current benchmark implementations do not scale to the complete quota of 60 cores. Data was averaged across one hundred separate runs.

For the `kronecker` matrix multiplication benchmark we explcitly turned off memory caching. With caching enabled, we observed severe performance degradation at 30 cores and beyond. We believe that the cause of this degradation stems from *false-sharing* effects; as concurrency increases across the chip, false-sharing becomes worse.



(a) L1 Read Miss (b) L2 Remote Read Miss

Figure 5: Cache Scaling Behavior for the `kronecker` Benchmark with Caching On

To support this claim we collected cache events through the OProfile tool provided with the Tilera development environment. The cache performance results are shown in Fig-

ure 5. Note that each sample corresponds to 500K L1 misses in Figure 5a and 50K L2 remote misses Figure 5b. This data indicates a significant increase in both L1 local and L2 remote read misses (resulting from invalidations by other cache line copies) which causes a reduction in memory read performance. We observed very few local L2 read misses in our experiments.

Finally, we also observed that the "performance gap" between different scheduling schemes generally increases with the number of cores. Figure 6 gives scaling data for the `indepwork` benchmark. The increasing variation of speedup across schemes as the number of cores increases is clearly identifiable.

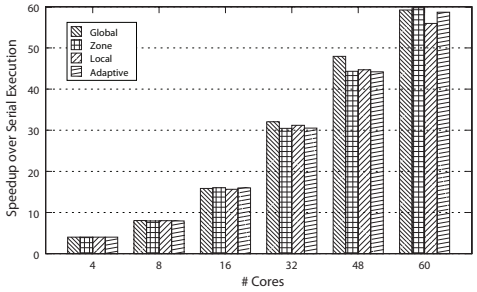


Figure 6: Performance Scaling of `indepwork` with Different Queueing Schemes

4.1.1 *Performance of Different Queuing Schemes*

To measure the performance of different queuing schemes, measurements were taken for each of the four queuing schemes; global queue (**G**), zone-queues (**Z**), local-queues (**L**) and the adaptive scheme (**A**). Data was normalized against the global (**G**) scheme.

The underlying queue implementations for all of the schemes is based on the lock-free queue scheme developed by Micheal et al. [11]. This scheme provides concurrent enqueue and dequeue operations on a FIFO queue through the use of compare-and-swap (CAS) operations on head and tail pointers. We chose the CAS-based lock-free queue even though the TILEPro64 processors only supports Test-and-Set H/W instructions; CAS operations are implemented through kernel-level fast mutexes. Although scaling emulated-CAS to hundreds of cores is possibly questionable, data for 60-cores shows good scaling of the independent work (`indepwork`) benchmark(see Figure 7) - the lock-free queue consistently outperforms the spinlock protected queue. Furthermore, the next-generation of TILE processors (the GX-series) are known to support H/W level CAS.

The experimental data, given in Figure 8, indicates a mixed result. The zone queuing scheme performed best in 3 out of the 6 tests. This scheme was most effective in the `fft` benchmark giving a 4% improvement over the next best scheme. The adaptive scheme gave a 5% and 2% improvement over the next best scheme for the `sort` and `segment` benchmarks respectively. Finally, the global scheme performed best for the `fibonacci` benchmark, beating the next-best scheme by 7%.

On average the worst performing scheme was `local` queuing - the scheme found in many existing lightweight thread based technologies such as Cilk [4] and OpenMP [13].

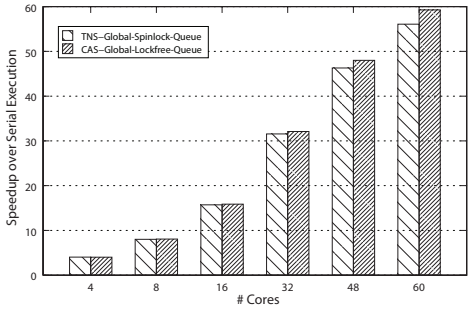


Figure 7: Test-and-set Spinlock Queue vs. Compare-and-Swap Lock-free Queue

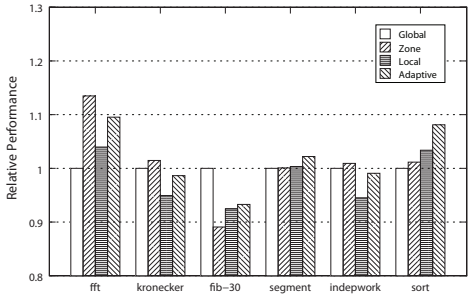


Figure 8: Benchmark Performance of Different Queuing Schemes

4.2 Throughput of Different Queuing Schemes

Our final set of experiments examined the throughput scalability of the different queuing schemes. For this we chose to use a synthetic benchmark in order to eliminate effects of inherent serialization in the benchmark application (as suggested by Amdahl's law) as well as potentially significant degradation caused by data locality and associated cache behavior as described in section 4.1. The synthetic benchmark "stress tests" task management and performs no real work. The test recursively creates tasks that perform a local-memory arithmetic calculation ($\sim$ 880K cycles). For this particular experiment, each task is configured to recursively create 10 children to a depth of 6 from the root node. A total of 2.2M tasks are created; task throughput is calculated over total execution time.

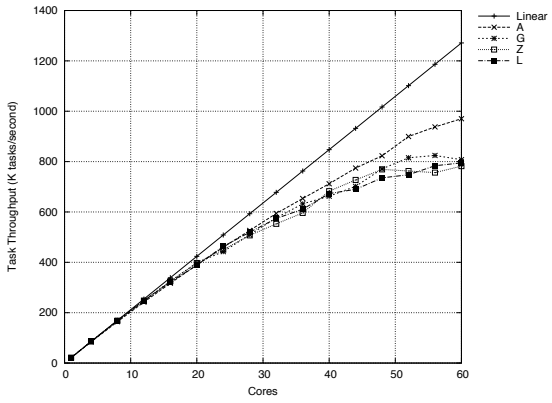


Figure 9: Scaling of Different Queuing Schemes

The results, given in Figure 9, show that the adaptive queuing scheme scales marginally better than the other schemes measured. However, at 60 cores, even this scheme degrades by 35% with respect to total task throughput, suggesting that scaling issues still exist. Nevertheless, we do consider this benchmark to present the extreme case with a high task throughput not typically found in useful applications.

5. RELATED WORK

Many of the recent improvements in C/C++ based parallel programming systems have been realized with lightweight user-level threading such as in Cilk [4], Intel's TBB [14], and Apple's GCD [1]. Most of these use work-stealing across multiple queues as the basis for their task management. The earliest implementation of work-stealing at least dates back to Halstead's MultiLisp [10] and Mohr et al.'s Lazy task creation [12]. Since then, the heuristic benefits of work-stealing with respect to space and communication has been well studied. Blumofe et al. presented a provably efficient randomized work-stealing algorithm for scheduling *fully strict* multithreaded computations [5], which was implemented in Cilk [8]. X10 work-stealing [2] later extended the Cilk algorithm to support *terminally strict* computations among other extensions. Guo et al. [9] introduce a *help-first* work-stealing scheduler, with compiler support for async-finish task parallelism. They present scenarios where help-first scheduling can yield better results than X10 style *work-first* execution.

The alternative to work-stealing is work-sharing, where whenever a new task is created, it is eagerly moved to an underutilized processor. Eager et al. [7] argue that work-sharing outperforms work-stealing at light to moderately loaded systems and when the cost of sharing is comparable with the cost of computation. While Guo et al. [9] compare the work-sharing scheduler in the open source X10 implementation [3] against their work-stealing policies and show that work-stealing generally performs better. In our experiments, we observe that work-sharing through global queuing scheme can in fact perform better in certain scenarios. Thus, we believe that no one scheduling policy can fit all the requirements.

6. CONCLUSION

Our results show that MIMD architecture-aware zone and adaptive queuing schemes can bring marginal perform advantages over the more commonly used local queuing scheme. Whilst the zone queuing scheme proved to be the best performer in the current benchmarks, the adaptive scheme shows promise for applications that have "stages" of different behavior with respect to concurrency and data sharing. Even though the measured performance advantage of the adaptive scheme is relatively small in these benchmarks, we believe that the benefit would increase for more complex applications and also for applications deployed across a distributed system context whereby the cost of task migration is very high.

This work has also provided a better understanding of building applications that can scale to larger manycore processors such as the TILEPro64. There are three key components to the successful MIMD application development. These are, i.) providing easy-to-use programming abstractions and achieving sufficient degree of task concurrency, ii.) minimizing contention on shared resources, and iii.) maintaining shared memory scalability by distributing tasks according to spatial locality. Our work thus far has primarily focused on the first two of these components. The current implementation of the SNAPPLE compiler does not perform any "intelligent" arrangement of tasks according to data relationships across lightweight tasks; this is a focus of continuing work.

7. REFERENCES

[1] Grand Central Dispatch (GCD) Reference, May 2010.

[2] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. A. Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA*, pages 229–240, 2007.

[3] R. Barik, V. Cave, C. Donawa, A. Kielstra, I. Peshansky, and V. Sarkar. Experiences with an SMP implementation for X10 based on the Java Concurrency Utilities. In *PMUP*, 2006.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. PPOPP, pages 207–216, 1995.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[6] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, 1986.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

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

[10] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Symposium on LISP and functional programming*, pages 9–17, 1984.

[11] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[12] E. Mohr, D. A. Kranz, R. H. Halstead, and Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2, 1991.

[13] OpenMP Architecture Review Board. Openmp application program interface, 2008.

[14] C. Pheatt. Intel threading building blocks. *J. Comput. Small Coll.*, 23:298–298, 2008.

[15] D. J. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. LCPC, pages 383–394, 2003.