On the Role of Deterministic Fine-Grain Data Synchronization for Scientific Applications: A Revisit in the Emerging Many-Core Era

Weirong Zhu, Ziang Hu, and Guang R. Gao
Department of Electrical & Computer Engineering
University of Delaware
Newark, Delaware 19716, USA
{weirong, hu, ggao}@capsl.udel.edu

Abstract

The design of microprocessor chip for high-end computing systems is moving towards many-core architectures with 10s or 100+ processing units. An important class of the target applications for such architectures are scientific numerical computations, many of which are intrinsically deterministic - that is for a given input a fixed output (result) should be produced no matter how the program is parallelized. It is critical that the read-after-write data dependencies in such programs should be implemented correctly and efficiently via fine-grain data synchronization. In this paper, we investigate the parallelization of three representative scientific computation kernels using fine-grain data synchronization supported by an recently proposed architectural mechanism for many-core chips, called Synchronization State Buffer (SSB) *[25]. Using detailed simulation on a simulator for the IBM 160-core Cyclops-64 chip architecture with the SSB extension, our experiments demonstrate significant performance advantage of using fine-grain data synchronization based parallelization schemes for scientific workloads.*

1 Introduction

Many-core architectures that integrate 10s (or beyond) of tightly-coupled processing cores on a single chip is emerging [10, 8]. In order to fully utilize the massive intra-chip parallelism provided by such chips, it is important to exploit the fine-grain parallelism inherent in applications. Efficient fine-grain synchronization is essential for determining the granularity of parallelism that can be exploited. For shared-memory multithreading programming model, which is normally employed for many-core architectures, synchronization ensures the correctness of the parallel programs by enforcing *mutual exclusion* and *read-after-write data dependence*. The reset of this paper will only address fine-grain synchronization that enforces read-after-write data dependencies among multiple concurrent threads. We will use the term *fine-grain data synchronization* to refer this type of synchronization throughout the paper.

An important class of the target applications for many-core architectures are scientific numerical computations, many of which are intrinsically deterministic - that is for a given input a fixed output (result) should be produced no matter how the program is parallelized. It is critical that the data dependencies in such programs should be realized efficiently to best exploit parallelism.

In [25], we have proposed a novel architectural mechanism to support fine-grain synchronization on many-core chips, called *Synchronization State Buffer (SSB)*. SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of active synchronized data units to support and accelerate word-level fine-grain synchronization, including both mutual exclusion and read-after-write data dependency synchronization.

One of the functionalities of SSB is to provide efficient fine-grain data synchronization, which ensures that a consumer thread reads a value at word-level in memory only after it has been written by a producer thread. Based on SSB, this paper investigates the parallelization of three representative scientific computation kernels. For each kernel, we demonstrate how it can be effectively parallelized with word-level fine-grain data synchronization.

To illustrate and understand the performance advantage of fine-grain synchronization based parallelization schemes, we conducted the experiments on a state-of-the-art many-core architecture – the 160-core IBM Cyclops-64 (C64) chip architecture [13]. We extended the C64 architecture simulator with the SSB architectural feature. Using detailed simulation, our experimental results demonstrate: (1) on multithreaded many-core architectures, fine-grain data syn-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

chronization mechanism is important and effective for exploiting fine-grain parallelism in scientific application kernels; (2) for many-core architectures, fine-grain synchronization based parallelization schemes can achieve significant performance improvement over the coarse-grain ones. For the three representative kernels we investigated, when running with 128 threads, fine-grain based implementation outperforms the coarse-grain ones by 38.1%, 312%, and 94.9% respectively; (3) with only modest hardware extension to many-core architectures, SSB provides an efficient mechanism for enforcing read-after-write data dependencies at word-level in memory among concurrent threads; and (4) in SSB implementation, a small buffer for each memory bank is sufficient for multithreaded scientific applications using fine-grain data synchronization.

2 SSB: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures

A new hardware architectural feature, called *Synchronization State Buffer (SSB)*, is recently proposed for many-core chips to support efficient word-level fine-grain synchronization [25]. Unlike the full/empty bits like fine-grain synchronization [21, 3, 11, 2, 17, 15], which tags the entire memory of the machine by associating additional access state bits with each word in memory, the design of SSB is motivated by the following observation: *at any instance only a small fraction of memory locations is actively participating in synchronization* [25].

Based on this observation, SSB is proposed as a modest hardware extension to many-core architectures. SSB is a buffer with a small number of entries attached to the memory controller of each memory bank. It records and manages states of *active synchronized data units* to support and accelerate word-level fine-grain synchronization. SSB caches the states of memory locations that are currently accessed by SSB synchronization operations. An interesting aspect of the SSB design is that it avoids enormous memory storage cost, and still creates an illusion that each word in memory is associated with a set of states that can be used to support word-level fine-grain synchronization. In case that an SSB for a memory bank is full, the SSB operation is trapped to software, which will apparently slows down the requested synchronization operation. However, it has been shown that the hardware synchronization resource provided by SSB is normally sufficient [25].

Because of the relatively smaller storage cost, each SSB entry can afford to encode large states – thus can support a rich set of synchronization functionalities. In the current design, SSB can be used to enforce mutual exclusion with various type of locks and read-after-write data dependencies between threads [25]. In the context of this paper, we are only interested in the fine-grain data synchronization.

Using SSB fine-grain synchronization operation is memory efficient. First, since SSB maintains the states for the synchronized memory locations in hardware, there is no need to allocate corresponding software-managed synchronization variables, which cost extra memory. Second, with one memory transaction, an SSB instruction does not only perform the synchronization on the memory location, but also brings the datum to the processor upon success. Therefore, a successful SSB operation combines synchronization and memory load into a single memory transaction.

The details of the principles, designs, and preliminary experimental results of SSB are described in the accompanying paper [25].

3 Case Studies in Parallelization with SSB Fine-Grain Synchronization

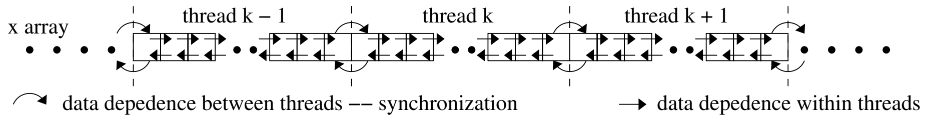
In this section, we investigate how to use fine-grain data synchronization to exploit fine-grain parallelism in scientific workload via case studies on three kernels. These three kernels are significant because they represent typical computation patterns when parallelizing scientific applications: the *1D Laplace solver* represents the iterative style of computation, the *linear recurrence equations* demonstrate irregular pattern of data dependencies, and the *wavefront computation* shows a general form of wavefront-like propagated computation in the solution space.

We attempt to parallelize these kernels using fine-grain data synchronization. Unlike global synchronization (i.e., barrier) based coarse-grain parallelization, where read-after-write data dependencies are enforced by making all consumers wait for all producers at a common synchronization point, the fine-grain data synchronization takes a point-to-point approach, which allows the consumer only waits for the data it needs for proceeding the computation. Therefore, fine-grain synchronization can avoid unnecessary waiting and global communication that caused by coarse-grain barrier synchronization.

3.1 1D Laplace Solver

Laplace's equation is a famous partial differential equation, which is important in many fields of science, such as electromagnetism, astronomy, and fluid dynamics. The 1D Laplace solver use a finite difference method to achieve numerical approximation of the equation, whose pseudo code is shown as follows:

```
for( i = 0; i < ITERATIONS; i++){
  for( j=1; j< TOTALSIZE-1; j++ )
    xnew[j] = 0.5*(x[(j-1)]+x[(j+1)]+b[j]);
    for( j=1; j< TOTALSIZE-1; j++)
      x[j] = xnew[j];
}
```



x array
thread k – 1
thread k
thread k + 1
data depedence between threads –– synchronization
data depedence within threads

Figure 1. Data Dependencies and Synchronizations in 1D Laplace Solver

In the kernel, at each iteration, every position of a 1D array is updated with a value function of its left and right neighbors that computed from the previous iteration. All the elements of the array need to be updated before the next iteration starts. For simplicity, within each iteration, two arrays are actually used. One stores the data computed by previous iteration, the other stores the data generated by the current iteration.

The multithreaded parallel implementation partitions the 1D array among threads, such that each partition only contains consecutive elements. To enforce the producer-consumer relation, a barrier is performed after all $xnew$ are computed, and another barrier is executed after $xnew$ is copied to x. This *barrier* based coarse-grain synchronization scheme enforces each thread to wait for all others completing the current iteration before starting the next one.

From the point of view of a thread, however, it only needs to wait for its two neighbor threads to supply the data at the border of its partition in order to continue its own computation (see Figure 1). Assuming that the portion of the x array assigned to a thread is between x_{start} and x_{end}, in order to start its next iteration, this thread only needs to read two elements from its two neighbors. For instance, for starting the computation of $xnew_{start}$ and $xnew_{end}$ at iteration i, the thread only needs its two neighbors to write their results into $x_{start-1}$, and x_{end+1} at iteration $i-1$.

Using this scheme, we can implement another parallel version of the solver using the SSB single-writer-single-reader operations to perform the fine-grain data synchronization between threads. The coarse-grain barriers are removed, the data synchronization is used to enforce each thread to wait for the data that is exactly necessary for starting the new iteration.

3.2 Linear Recurrence Equations

Linear recurrence equations are widely used in scientific linear algebra computations. Livermore Loop 6 [14] represents a general form of linear recurrence equations:

```
for ( i=1 ; i<n ; i++ )
   for ( k=0 ; k<i ; k++ )
      W[i] += b[k][i] * W[(i-k)-1];
```

As shown in Figure 2, the outer loop computes the array `W`, and at each iteration `i`, `W[i]` depends on values computed in all previous iterations, that is, `W[i]` depends on

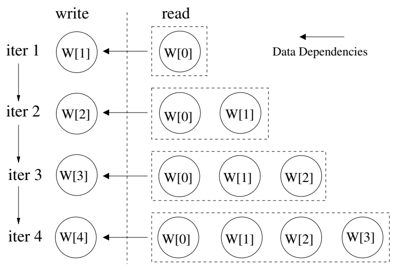


Figure 2. Characteristics of Linear Recurrence Equations

`W[0]`, `W[1]`, ... , `W[i-1]`. Such cross-iteration dependencies of array `W` makes it difficult to parallelize this loop at compilation time [23].

An iteration of the outer loop is an inner product of know values. Given *infinite* number of threads, the time complexity is $O(\log n)$ [14]. Observe that an element $W[i]$ appears in the sum of all elements whose indices are higher than its, i.e., $W[i{+}1]$, $W[i{+}2]$, ... , and $W[n]$. Therefore, at iteration i instead of summing elements $W[1]$ through $W[i-1]$ to compute $W[i]$, one can add $W[i-1]$ multiplied by the corresponding elements of b to $W[i]$ through $W[n]$ [14]. The loop after such transformation is shown as follows:

```
for ( k=0 ; k< n - 1 ; k++ )
   for ( i= k + 1 ; i < n ; i++ )
      W[i] += b[i-1][i-k] * W[k];
```

In the new loop, at an iteration of the outer loop, all the addition in the inner loop can be done in parallel, thus the time complexity is $O(1)$ if infinite number of threads is assumed [14]. Our coarse-grain implementation is based on the new loop. At each iteration of the outer loop, the computation of the inner loop is partitioned to different threads. After the computation, all threads join a barrier, then start next iteration.

However, we regard the barrier-based coarse-grain approach has several drawbacks. First, The parallelism at each iteration (outer loop) is continuously decreasing along the computation. Second, since the number of iterations of the inner loop, which keeps changing, can not always be divided evenly among threads, the computation is normally unbalanced. Third, notice that each iteration of the inner loop computes a different $W[i]$. When an iteration of the inner loop is assigned to a thread, a corresponding $W[i]$ is

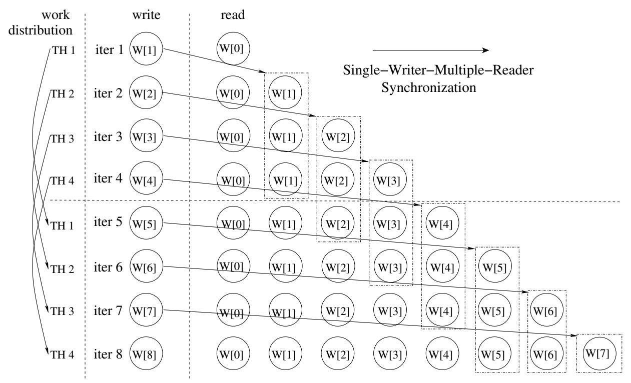


Figure 3. Parallelization and Synchronization of Livermore Loop 6 (*4 threads, round-robin scheduling, chunk size = 1***)**

loaded, a new value is computed, and stored back to the global memory. Therefore, there is no locality exploited for $W[i]$. Moreover, the barrier at the end of each iteration creates an "all-to-all wait" scenario, which requires global communication among threads.

To address these issues, we use SSB-based fine-grain synchronization to parallelize the loop. The parallelization is based on the original loop code. For simplicity, we assign the iterations of the outer loop to different threads in a round-robin fashion. Using this scheduling policy, the computation for a $W[i]$ is performed by a single thread. Therefore, the partial result of $W[i]$ can be kept in the register of the thread to exploit the locality. Only when the final result is computed, the $W[i]$ is written back to global memory.

By analyzing the loop nests carefully, our actual parallelization and synchronization scheme is shown in Figure 3, which illustrates the case where 8 iterations are concurrently executed by 4 threads, and the chunk size of round-robin scheduling is 1 iteration. When thread 1 completes iteration 1, it notifies threads 2, 3, and 4 about the availability of $W[1]$. Thread 1 then executes iteration 5 according to the round-robin work distribution policy. Although the computation of iteration 5 depends on $W[1]$ to $W[4]$, it does not have to explicitly wait for $W[1]$, since thread 1 itself computed $W[1]$ previously. Similarly, when thread 2 moves to iteration 6, it does not need to check the availability of $W[1]$, or $W[2]$, because $W[2]$ is computed by itself previously, and when $W[2]$ is available, $W[1]$ is ensured to be available. By taking this synchronization strategy, after the computation of an iteration, a thread performs a synchronized write to the memory to notify num_threads $- 1$ readers. When a thread begins a new iteration i to compute $W[i]$, it uses normal load operations to read from $W[0]$ to $W[(i-1)-(\text{num\_threads}-1)]$, and uses synchronized read to load the remaining num_threads $- 1$ elements of W. As a result, no matter how large the problem size is, the number of synchronization reads and writes required only depends on the number of threads.

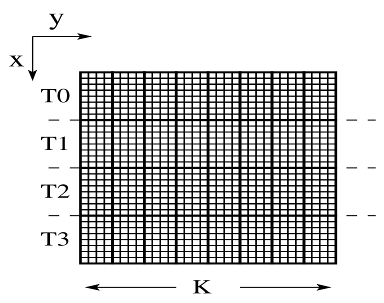
Compared to the barrier-based coarse-grain approach, our fine-grain solution 1) exploits the inherent fine-grain parallelism in the computation, thus can achieve better workload balancing at runtime; 2) achieves much better locality as explained before; and 3) eliminates the use of barrier, thus avoids the overhead of the barrier as well as the unnecessary "all-to-all wait" scenario.

3.3 Wavefront Computation

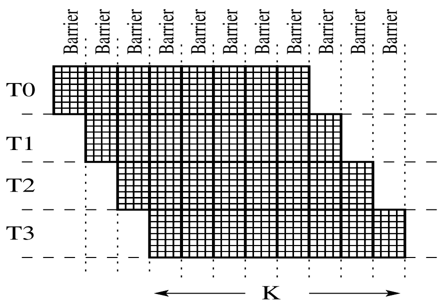
Wavefront computations are common in scientific applications. In this paper, we focus on the 2D wavefront computation problem. Given a matrix, the left and top edges of which are all initialized, the computation of each remaining element depends on its neighbors to the left, above, and above-left. If the solution is computed in parallel, the computation at any instant form a wavefront propagating forwards in the solution space.

Figure 4 illustrates the coarse-grain parallelization scheme inspired by [24]. The solution space is partitioned along the x dimension. Let T be the number of threads, X be the number of rows. Each thread is assigned with the computation of X/T contiguous rows. In order to gain parallelism, the solution space is further partitioned to K blocks along the y dimensions. Each thread completes all the computation in a block, joins a barrier, and starts the computation of the next block, as shown in Figure 4(b). The parameter K determines the degree of the parallelism. With the increase of K, the granularity of data associated with each barrier synchronization is decreasing, and the number of global synchronizations (barriers) required is increasing. Therefore, the level of parallelism that can be exploited is determined by the efficiency of the barrier synchronization.

In our fine-grain implementation, the rows of the matrix are assigned to threads in a round-robin fashion (modulo T). With this static scheduling policy, to compute an element, only the availability of its above neighbor needs to be checked. SSB fine-grain single-writer-single-reader synchronization can be used to enforce the data dependencies. A straightforward synchronization scheme is to allow synchronized read/write on each data elements. To reduce the amount of synchronization, we group 8 consecutive elements in a row as a block. Once a thread completes the computation for a block, it writes the first element of the block to the memory with a synchronized write, and the other elements in the block are written with normal store instructions. Afterwards the thread moves to the next block. Before the computation of a block, a thread performs a synchronized read to get the first element of the block, the remaining elements of the block are read with normal



(a) Partition the Solution Space



(b) Computation Scheme in Parallel

Figure 4. Coarse-Grain Parallelization of Wavefront Computation: Number of Threads $T = 4$, Number of Computation Blocks $K = 2 * T = 8$.

load instructions. With the fine-grain solution, although the computation is still in a wavefront form, a thread can be kept busy as soon as the block, which it is waiting for, becomes available. Except the prologue and epilogue stages of the computation, all threads can be kept usefully busy in a pipelined fashion. Unlike the global synchronization with barrier, threads never wait unnecessarily.

4 Evaluation Framework

We evaluate the performance of the three kernels on a state-of-the-art many-core architecture – the IBM 160-core Cyclops-64 (C64) chip architecture [13].

The C64 chip architecture explores a many-core design by integrating a large number (160) of hardware thread units and embedded SRAM memory banks (32KB each) on a single silicon die. C64 also employs an explicitly addressable memory hierarchy without data cache. All the on-chip and off-chip resources (e.g., 4 off-chip DRAM banks, 250MB each) are connected through a high-bandwidth crossbar network. The crossbar also guarantees a sequential consistency memory model for the C64 chip architecture [13]. C64 has efficient support for thread level execution, such as ISA-level sleep/wakeup instructions. All the thread units within a C64 chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers. It is worth noting that we use the hardware barrier in the implementation of the coarse-grain parallelization approaches for the three kernels. C64 provides no hardware support for context switch, and uses a non-preemptive thread execution model. The peak performance of a C64 chip is 80GFLOPS.

We implemented SSB as an extension to the C64 ISA using an execution-driven binary-compatible simulator for the C64 many-core architecture [12]. We model the C64 chip design with the 160 cores, the multi-level memory hierarchy, and the crossbar interconnection network. The simulator takes into account the main sources of pipeline delays and stalls in the processor architecture, as well as models all details in the memory hierarchy, including contention in memory and the crossbar network. The SSB extension to C64 is implemented in the simulator. SSB instructions that require return values have the same latency as a load instruction, otherwise the latency is same as a store instruction. For our experiments, the chosen size of SSB for on-chip and off-chip banks are 16 and 1,024 respectively. Both are 8-way set associative.

5 Experimental Results

In this section, we demonstrate the efficiency and effectiveness of fine-grain data synchronization through the experimental results.

5.1 1D Laplace Solver

SSB-based fine-grain synchronization naturally expresses the data dependencies in the 1D Laplace solver. The "one-to-one wait" data synchronization strategy avoids the unnecessary "all-to-all wait" scenario due to the use of barrier as well as the overhead of barrier. As a result, the SSB-based fine-grain synchronization approach beats the barrier based coarse-grain counterpart in all cases, even the C64 hardware-based barrier is very efficient. For example, when the solver runs on 128 threads with a problem size of 4,096, the SSB-based version can achieve a speed up of 109, and outperform the coarse-grain version by 38.1%.

5.2 Linear Recurrence Equations

Given a W array with size 5,120, Figure 6 compares the fine-grain data synchronization based approach with the coarse-grain based one. For the fine-grain approach, the *chunk_size*, is the number of iterations to be scheduled per

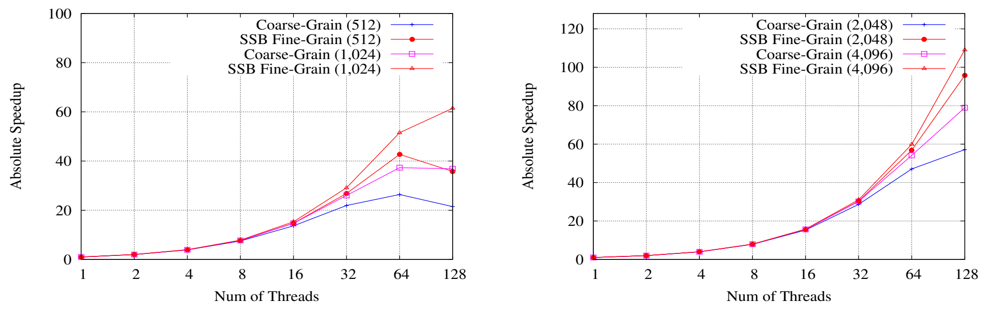


Figure 5. Barrier-based Coarse-Grain Synchronization vs. SSB-based Fine-Grain Synchronization for 1D Laplace Solver. *Problem size: 512, 1,024, 2,048, 4,096*

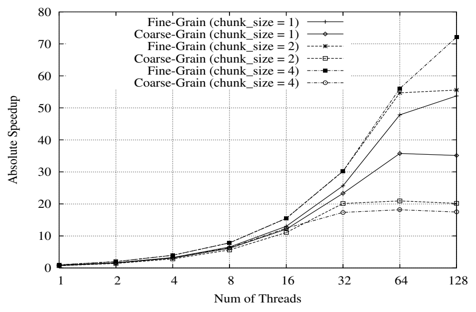


Figure 6. Speedup of Livermore Loop 6

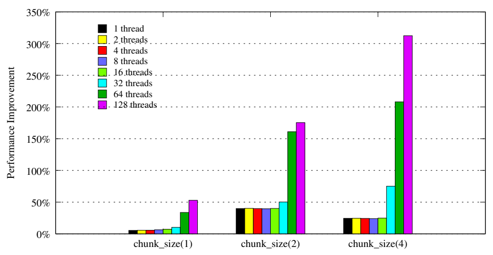


Figure 7. Livermore Loop 6: Fine-Grain vs Coarse-Grain

time by the round-robin algorithm. For the coarse-grain approach, the parallel version is based on a sequential version that has been loop unrolled certain times specified by the *chunk_size*. In Figure 6, when *chunk_size* equals to 2 or 4, the speedups are calculated against the sequential versions, which have also been loop unrolled twice and 4 times respectively. Therefore, the comparison of two curves will be meaningful, only if the *chunk_size* is the same. As shown in Figure 6, the fine-grain data synchronization based approach always performs better when running on a large number of threads.

Figure 7 shows the performance improvement of the SSB-based fine-grain approach over the coarse-grain one (calculated as $(Speedup_{fine-grain} - Speedup_{coarse-grain})/Speedup_{coarse-grain})$. We can observe that the performance improvement increases significantly when the number of threads is large. For example, when 128 threads are used, the fine-grained approach with a chunk size of 4 achieves an absolute speedup of 72, which demonstrates a 312% improvement over the corresponding coarse-grained one. The performance advantage of the SSB-based fine-grain approach is attributed to 1) fine-grain parallelism exploited by SSB-based synchronization solution; 2) better data locality; and 3) the removal of the barriers.

5.3 Wavefront Computation

Our experiments are conducted with a 1024×1024 matrix [1]. Figure 8 compares the speedups of the fine-grain approach to the coarse-grain ones.

Recall that we group 8 consecutive elements in a row as a block in our fine-grain synchronization based parallelization scheme. In the code, the computation of the 8 elements in a block is written in a similar way as loop unrolling. To make a fair comparison, the innermost loop of the coarse-grain based implementation is also unrolled 8 times. The speedups shown in Figure 8 is also calculated against the sequential version, the inner loop of which has been unrolled 8 times. For the coarse-grain approach, we experiment with three different K values.

From Figure 8, it is apparent that the SSB-based fine-grain approach outperforms others when running with multiple threads. For the coarse-grain versions, it can also be observed that a larger K can improve performance only if the number of threads (T) is moderate. When number of threads is large, the cost of barrier cancels out the benefit

[1] A 1024×1024 matrix of doubles exceeds the capacity of on-chip SRAM memory of current C64 chip design. For the purpose of our experiments, we extend the on-chip SRAM memory to 10M in the simulator.

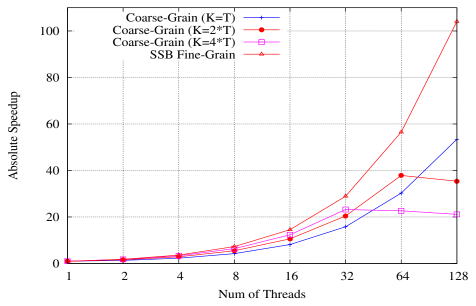


Figure 8. Speedup of Wavefront Computation

brought by associating finer grain of data (due to larger K) with each barrier synchronization.

Although the data dependencies in wavefront computation implies serialization, the multithreaded implementation with fine-grain data synchronization demonstrates the capability to exploit the inherent parallelism within such computation form. When running with 128 threads, the SSB-based implementation shows a speedup of 104, which outperforms the three coarse-grain implementations by 94.9%, 194.2%, and 392.7%, respectively.

5.4 Effectiveness of Fine-Grain Synchronization

An SSB synchronized write fails only if the previous synchronized write has not been consumed by a synchronized read, while an SSB synchronized read fails if the target data is not available yet. Both experiences a slow down if the corresponding SSB happens to be full, since the synchronization operation is trapped to software. A successful SSB synchronized read can combine synchronization and memory load into a single memory transaction, thus incurs low overhead compared to a normal load operation. Therefore, successful synchronization is always desired.

Benchmark	64 threads		128 threads	
	Success Rate	Full Rate	Success Rate	Full Rate
1D Laplace Solver (4,096)	88.20%	0	84.29%	0
Livermore Loop6 (chunk size: 4)	87.52%	0	72.13%	0
Wavefront	99.86%	0	99.83%	0

Table 1. Synchronization Success Rates and SSB Full Rates

We measured the percentage of successful synchronizations among all synchronizations issued for the three kernels as shown in Table 1. We can see that even for large number of threads, most fine-grain synchronization operations are successful, which in turn ensures low cost of synchronization. We also observed that the SSB buffer never fills up for the the three kernels. This analysis shows that a small SSB for each memory bank is normally sufficient to cache the access states of outstanding synchronizing data units for multithreading programs. Using modest hardware cost, SSB achieves the same effect as if each word of the entire memory is tagged.

6 Related Work

HEP [21], Tera [3], MDP [11], Alewife [18, 2], M-Machine [17], Cray MTA-2 [1], the MT processor in Eldorado [15], and others associate additional access state bits (e.g., *full/empty bits*) with each word in entire memory. Fine-grain synchronization is achieved by accessing those word-level state bits in memory. Experience of parallelizing scientific code with full/empty bits fine-grain synchronization mechanism on Tera [3] or its successor Cray MTA-2 [1] has been reported in many literatures [22, 9, 5]. Agarwal et. al have reported their experience on the MIT Alewife machine [2]. They evaluated the performance of scientific applications, such as SOR, and MICCG3D, parallelized using J-structure and L-structure supported by hardware full/empty bits [18].

An *I-structure* is a data structure proposed to facilitate parallel computing [6] on dataflow model based systems. Using single assignment semantics. an I-structure element can only be written once, but it can be read many times. Producer-consumer type of fine-grain data synchronization is achieved by interacting with the state of an I-structure when accessing it. Unlike I-structure, which regards the redefinition of an element as an error, the *M-structure* is a fully mutable data structure such that an element can be redefined repeatedly [7].

Recently, hardware transactional memory (TM) [16, 4, 20, 19], is proposed to support non-blocking fine-grain synchronization. A transaction is a sequence of memory reads and writes executed by a single thread, which is guaranteed to be atomic and serializable. TM systems provide great potential to facilitate multithreading programming. The design of TM intends to provide a non-blocking replacement for the lock-based mutual exclusion synchronization. SSB does not only support fine-grain mutual exclusion, in a blocking fashion, but also fully support deterministic fine-grain data synchronization that enforces read-after-write data dependencies among concurrent threads.

7 Summary

In this paper, we reported our experience on parallelization of three representative scientific application kernels on a multithreaded many-core architectures. Using fine-grain data synchronization mechanisms supported by SSB, a recently proposed architectural extension to many-core chips, we illustrate how the three kernels can be parallelized to exploit fine-grain parallelism by avoiding unnecessary waiting and global communication due to the use of coarse-grain synchronization. By experimenting on the simulator for the 160-core IBM Cyclops-64 architecture, significant performance benefit from using fine-grain data synchronization has been observed.

Acknowledgement

We would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of the IBM Cyclops-64 architecture, ETI, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. We would also like to acknowledge other members of the CAPSL group at University of Delaware, in particular Ioannis Venetis, and Juan del Cuvillo. We would also like to thank Vugranam Sreedhar from IBM for many useful discussions.

References

[1] Cray MTA-2 System.

[2] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeoung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. *SIGARCH Comput. Archit. News*, 18(3b):1–6, 1990.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Procs. of the 11th Intl. Symp. on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[5] W. Anderson, P. Briggs, C. Hellberg, D. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experience with scientific programs on the Cray MTA-2. *Procs. of the ACM/IEEE SC2003 Conf.*, 2003.

[6] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

[7] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *in Proc. of Conf. on 1991 Functional Programming Languages and Computer Architectures*, pages 538–568, 1991.

[8] S. Y. Borkar, H. Mulder, P. Dubey, S. S. Pawlowski, K. C. Kahn, J. R. Rattner, and D. J. Kuck. Platform 2015: Intel processor and platform evolution for the next decade, 2005.

[9] L. Carter, J. Feo, and A. Snavely. Performance and programming experience on the Tera MTA, 1999.

[10] B. Dally. Computer architecture in the many-core era. In *Key note in the 24th Intl. Conf. on Comput. Design*, 2006.

[11] W. J. Dally and et. al. The message-driven processor. *IEEE Micro.*, 12(2):23–39, 1992.

[12] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS2005), in conjuction with ISCA2005*, Madison, Wisconsin, June 2005.

[13] M. Denneau and H. S. Warren, Jr. 64-bit Cyclops principles of operation, Apr. 2005.

[14] J. Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing*, 7(2):163–185, 1988.

[15] J. Feo and et. al. Eldorado. In *Procs. of the 2nd Conf. on Computing Frontiers*, pages 28–34, 2005.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Procs. of the 20th Intl. Symp. on Computer Architecture*, 1993.

[17] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *the Procs. of 25th Intl. Symp. on Computer Architecture*, 1998.

[18] D. Kranz, B. H. Lim, and A. Agarwal. Low-cost support for fine-grain synchronization in multiprocessors. Technical Report MIT/LCS/TM-470, 1992.

[19] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *the Procs. of 33rd Intl. Symp. on Computer Architecture*, Washington, DC, 2006.

[20] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Procs of the 12th Intl. Symp. on High Performance Computer Architecture*, Feb. 2006.

[21] B. Smith. The architecture of HEP. In J. S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, Scientific Computation Series, pages 41–55. MIT Press, Cambridge, MA, 1985.

[22] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz. Multi-processor performance on the Tera MTA. In *Procs.of the 1998 ACM/IEEE Conf. on Supercomputing*, pages 1–8, 1998.

[23] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *the 5th Intl. Symp. on High-Performance Computer Architecture*, pages 54–58, Orlando, Florida, Jan. 9–13, 1999.

[24] D. Yeung and A. Agarwal. Experience with fine-grain synchronization in mimd machines for preconditioned conjugate gradient. In *Procs. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 187–192, 1993.

[25] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. *CAPSL Technical Memo 67 Revised*, Nov. 20th 2006.