A Comparison Of Shared Memory Parallel Programming Models

Jace A. Mogill *David J. Haglin*

Pacific Northwest National Laboratory
Richland, WA, USA
{jace.mogill, david.haglin}@pnl.gov

ABSTRACT:

The dominant parallel programming models for shared memory computers, Pthreads and OpenMP, are both *thread-centric* in that they are based on explicit management of tasks and enforce data dependencies and output ordering through task management. By comparison, the Cray XMT programming model is *data-centric* where the primary concern of the programmer is managing data dependencies, allowing threads to progress in a data flow fashion. The XMT implements this programming model by associating tag bits with each word of memory, affording efficient fine grained data synchronization independent of the number of processors or how tasks are scheduled. When task management is implicit and synchronization is abundant, efficient, and easy to use, programmers have viable alternatives to traditional thread-centric algorithms. In this paper we compare the amount of available parallelism relative to the amount of work in a variety of different algorithms and data structures when synchronization does not need to be rationed, as well as identify opportunities for platform and performance portability of the data-centric programming model on multi-core processors.

KEYWORDS: Programming Models, Cray XMT, OpenMP, Parallelism

1 Introduction

All parallel programming models for shared memory computers are comprised of two orthogonal concepts: the source of concurrency and the synchronization primitives used to manage the concurrency. These responsibilities may be implemented in hardware, software, or both, and typically may be mixed freely. At one end of the design space is the Cray XMT which implements as much of this functionality in hardware as possible, whereas conventional x86 multi-core microprocessors rely almost entirely on software. The XMT's rich and efficient hardware level concurrency and synchronization afford fine-grained parallel programming models such as *data flow* that are not feasible with software only. In this paper we describe how synchronization is much more than just syntax – efficient and easy-to-use synchronization of the XMT qualitatively changes the parallel programming model by increasing the amount of concurrency available for hardware to exploit.

1.1 Programming Models

The two dominant shared memory programming models are Pthreads and OpenMP. The Pthreads programming interface is specified in POSIX and is typically accessed via runtime library and operating system calls. Any ANSI/ISO-C conforming compiler may be used to compile programs which use Pthreads and the compiler has no direct knowledge of con-

currency in the program. OpenMP extends the programming language (C or Fortran) with directives to make parallelism and data privacy explicit. OpenMP annotations change the semantics of loops and data persistence and it is possible for OpenMP annotations to assert incorrect program transformations which manifest themselves as race conditions or deadlocks.

Because Pthreads are frequently implemented in the operating system and runtime, requiring no special compiler support, Pthreads are often used as the underlying infrastructure to implement OpenMP in compilers. Pthreads are, in fact, a superset of OpenMP's functionality, and many OpenMP implementations allow Pthreads and OpenMP to be mixed. The semantics of OpenMP pragmas are implemented and enforced by the compiler and the OpenMP runtime library, meaning an OpenMP capable compiler is required to compile OpenMP annotated programs. Furthermore, the execution performance of the program is highly dependent on the quality of the OpenMP implementation.

Parallel programming on the XMT is unlike Pthreads and OpenMP in that it is largely implicit; in some circumstances, the programmer may be required to describe why a particular loop is parallelizable, but the programmer is not involved in describing how the parallelism should be implemented. The XMT compiler performs a static dependence analysis similar to that performed by automatic vectorizing compilers, and then automatically parallelizes loops that are proven to not have dependences between iterations. By comparison, OpenMP does not perform any analysis or any automatic parallelization, and Pthreads is just a library and has nothing to do with the compiler at all.

It is important to note that parallelism need not be represented by loop constructs in the source code. In OpenMP it is possible to introduce parallelism at an arbitrary place in the program, causing all the statements in the parallel region to be executed by each thread. Loops without any OpenMP annotation will execute every iteration on every thread, whereas OpenMP annotated loops inside an OpenMP parallel region are decomposed among the existing threads.

In Pthreads a new thread is created by an explicit call to the operating system specifying the entry point for the new thread and a single argument. To create multiple new threads for a parallel region, a loop creating one thread per iteration or a fanout tree is required. Similarly, at the end of a parallel region a loop or other reduction "joining" each thread is needed to ensure all threads have finished executing before proceeding.

Vector parallelism is sometimes referred to as "data parallelism" because there are no dependencies between any two elements of a vector, permitting the calculations on the vector to take place in any order (or concurrently). This constraint on data means vector parallelism never requires any kind of synchronization, making it easy for programmers and compilers to implement data parallelism correctly by analyzing static data dependencies that are explicit in the code.

The parallelism made possible by multiple CPU cores must also honor the constraint that a single piece of data not be operated on in two places simultaneously; otherwise a race condition will occur. However, unlike vector processors, multi-processors implement additional data semantics, which detect these conditions at execution time through cache coherency mechanisms and the processor automatically serializes operations so data races do not occur. This runtime dependence functionality comes at a steep performance cost as it is intended to manage exception cases, not to be relied on as the nominal programming idiom.

1.2 Historical Development

Shared memory parallel programming has been part of High Performance Computing (HPC) since the Cray X-MP was introduced in 1985 [1]. As a vector processor based machine, the X-MP was also an early example of a hybrid-microparallel computer: a lower level of communication-free parallelism in a single vector processor, and a higher level parallelism of multiple processors working concurrently. Long

vectors were often strip-mined to provide nested parallelism, but explicitly annotated outer loops were sometimes required in order to fully utilize the Cray X-MP.

Programming was usually performed in FORTRAN77 with a combination of implicit loop parallelism and explicit micro-tasking directives. Cray micro-tasking directives typically instructed the compiler to decompose the iterations of a single loop across multiple processors. The Multi-Streaming processors of the Cray X-1 were programmed in a similar fashion. SGI's multiprocessor RISC based systems has their own set of directives. Several lines of equivalent directives for these different compilers (figure 2) were a common sight prior to OpenMP.

When SGI bought Cray they found themselves with two different and incompatible sets of directives for parallel programming. Their solution was to form the OpenMP consortium to define a set of portable directives based on commonalities from the existing methods. The substantial overlap of semantics readily yielded a set of OpenMP directive which replaces the directive soup with a single directive seen in figure 3.

Since 2003, commodity x86 microprocessors have essentially combined the X-MP's hybrid system model onto a single chip: vector function units are implemented in the streaming single-instruction multiple-data (SSE and MMX) instructions, and the processor core is replicated several times per socket; a conventional microprocessor is indeed a Cray X-MP on a chip.

2 Expressing Synchronization and Parallelism

Regardless of the syntax of the parallel programming model, the semantics can be divided into two parts: The part(s) regarding where and to what degree the program should begin executing in parallel, and the part(s) regarding the flow control (synchronization) of the threads (Figure 4). In Pthreads all aspects of parallelism are explicit, and the parts can be identified by the particular Pthread function call. OpenMP explicitly declares parallel regions, but much of the synchronization is managed implicitly.

2.1 So Called Lock- and Wait-Free Algorithms

Developing algorithms and data structures (including hash tables) for lock-free components (see [2, 3]) has recently received attention. The definition of "lock-free" (sometimes called non-blocking or wait-free) in these algorithms is much more relaxed than "cannot use locks". To provide a "lock-free" service, the component must guarantee that whenever a thread executes some finite number of steps, at least one thread must make progress toward completing its task [4]. Using this definition it may be more appropriate to call these components *dead-lock free*.

The x86 instruction which implements a locked compare and swap is an obvious and frequent choice for implementations of these so called wait-free algorithms. Compare And Swap corresponds to the Pthreads primitive `pthread_mutex_trylock()` but this functionality is not available in OpenMP and must be accessed via compiler primitives or direct assembly language code. For each use of CAS there must be explicit success and fail outcomes, which is predicated on there being something else to do if the desired lock cannot be acquired. This begs several questions:

- If there is other work to do which does not require the lock, why wasn't it done first?
- If another lock will be tried, why not retry this same lock instead?
- What is the cost of issuing speculative work which will be discarded relative to the cost of just waiting?
- What is the probability that issuing speculative work does not interfere with work which will definitely make progress?
- If the algorithm must keep trying locks until it acquires one, how is this different from spinning?

The analysis of most lock-free or wait-free algorithms fail to include the cost of discarded

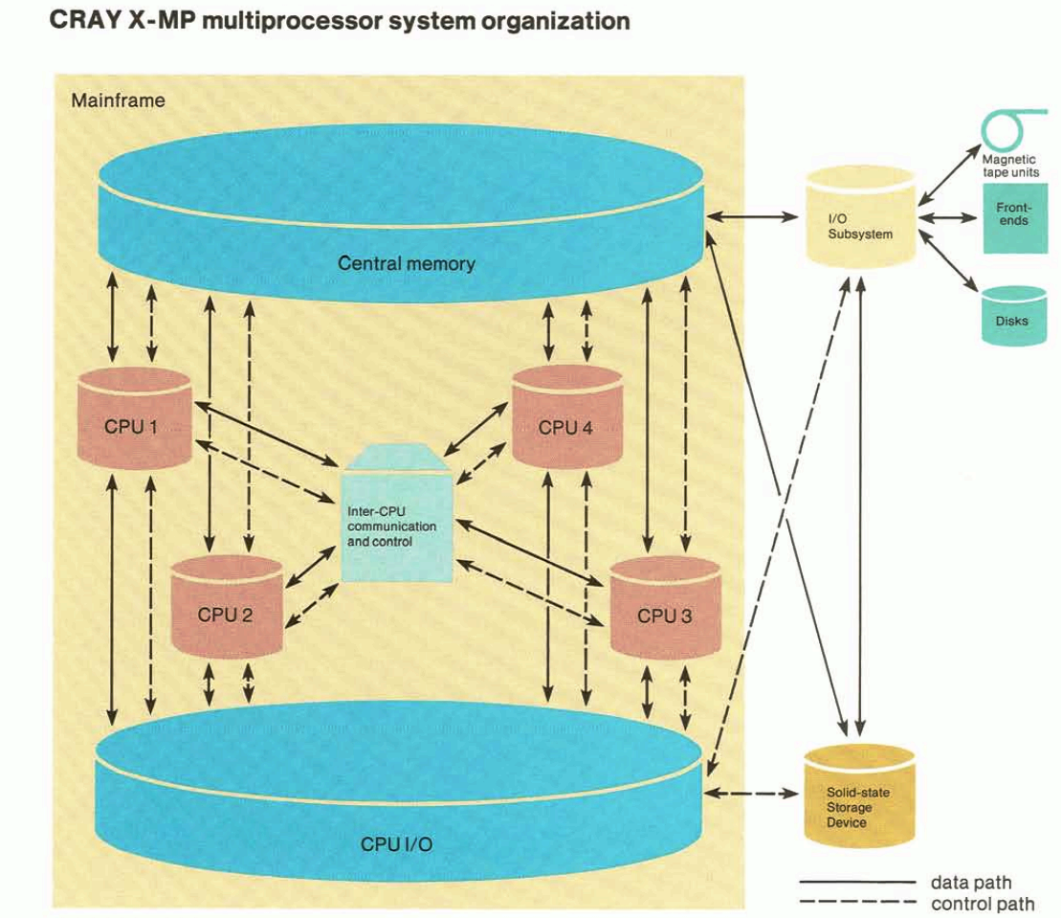

Figure 1: Block diagram of the Cray X-MP. A diagram of a multi-core x86 processor will look substantially similar, with "Inter-CPU Communication and Control" functionality being implemented in the memory controller.

```
!CSD$ PARALLEL DO PRIVATE(I)
CMIC$ DO ALL VECTOR SHARED(LENH, IADDH, IM, VBL, SCR) PRIVATE(I)
C$ DOACROSS SHARE(LENH, IADDH, IM, VBL, SCR) LOCAL(I)
      DO I=1,LENH
        SCR(IM+I,1)=.5E0*(VBL(IADDH+IM+I-1)+VBL(IADDH+2*IM+I-1))
        SCR(IM+I,2)=1.E0/SCR(IM+I,1)
      ENDDO
```

Figure 2: Cray Microtasking (`CMIC$`), Cray Multi-Streaming (`!CSD$`), and SGI (`C$`) directives decomposing a loop across multiple vector processors.

speculative work. Speculative work consumes data bandwidth, can pollute caches, steals instruction bandwidth from threads that could otherwise make progress, and consumes power adding to hardware cost and complexity. Ultimately, many lock-free algorithms are both algorithmically and power inefficient.

```
C$OMP PARALLEL DO DEFAULT(SHARED)
      DO I=1,LENH
        SCR(IM+I,1)=.5E0*(VBL(IADDH+IM+I-1)+VBL(IADDH+2*IM+I-1))
        SCR(IM+I,2)=1.E0/SCR(IM+I,1)
      ENDDO
```

Figure 3: OpenMP directive replacing Cray Microtasking, Cray Multi-Streaming, and SGI.

		Parallelism	
		Explicit	Implicit
Synchronization	Task/Thread	Pthreads/OpenMP	Vectorization
	Data	XMT	Data flow

Figure 4: Orthogonality of the source of parallelism and synchronization.

2.2 Synchronization on the x86 Architecture

No meta-data about the user's data is stored in the main memory of conventional computers, however caches, memory controllers and processors frequently store meta-data about where copies of data are and which copies are up-to-date with respect to other copies [5]. Synchronization and atomicity are enforced by memory access protocols and cache coherency which have access to the meta-data. The x86 instruction set offers a small number of instructions which are can be combined with a `LOCK#` prefix defined with extended memory semantics. The "Lock Signal" which is used to define the locked instructions is never directly exposed in the x86 ABI, allowing implementations to optimize or replace it with another mechanism entirely.

A partial list of locked x86 instructions:

- **xadd, xsub**: Atomic add, subtract
- **xand, xnot, xxor**: Bitwise operations
- **xchg**: Atomic swap
- **xcmpxchg**: Atomic compare and Swap

The atomic arithmetic and bitwise operations implement thread-safe versions of their ordinary counterparts. The atomic compare and swap instruction can be used to implement a mutual exclusion lock. A word of storage representing the lock is initialized to a value representing unlocked, and subsequent Compare-And-Swap instructions try to swap the unlocked value with their thread ID number. A successful swap indicates the mutual exclusion lock was acquired.

None of the atomic instructions permit any concurrency. To the contrary, their locked or atomic nature defines that there is no concurrency. The `LOCK#` prefix on atomic instructions is an explicit micro-architectural mutual exclusion region that works in the same way as a software mutual exclusion lock or critical region. Specifically, on systems which have an explicit data lock separate from a data operation, or implement this functionality in microcode, atomic (or locked) compare and swap is equivalent to:

```
lock(data)
if(data == compare value)
  data = new value
endif
unlock(data)
```

In situations where loop bodies are only a few instructions long, the number of hardware processors may be greater than the number of instructions in the loop body. In these instances some instructions must be executing in multiple processors simultaneously, and if those instructions includes any of the x86 locked instructions operating on shared data, the execution would serialize at that instruction. The fastest time

such a loop can be executed is $T = N$, the number of iterations, not the number of iterations divided by the number of processors. This scenario is far more common than one would first think, and will become more common as the number of cores increases over time.

It is important to note that cache coherency mechanisms are architected to handle exception cases, not efficiently manage the synchronization of data between hardware processors. The performance penalty for inefficient use of these features can be 100x or more slower than implementations with no false sharing or synchronization.

2.3 Synchronization on the Cray XMT

The Cray XMT is the commercial name for the shared-memory multithreaded machine developed by Cray under the code name "Eldorado" [6, 7]. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with *Threadstorm* processors. The entire system is connected using the Cray Seastar-2.2 high speed interconnect. The system we use in this study has 128 processors and 1 TB of shared memory.

Each Threadstorm processor is able to schedule 128 fine-grained hardware threads (the XMT terminology for this is *stream*) to avoid memory-access generated pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a hardware stream comprised of a program counter, a status word, 8 target registers and 32 general purpose registers. Each Threadstorm processor has a VLIW (Very Long Instruction Word) pipeline containing operations for the Memory functional unit, the Arithmetic unit and the Control unit.

Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grained thread synchronization with little overhead. The memory is hashed at a granularity of 64 bytes and fully accessible through load/store operations to any Threadstorm processor connected to the Seastar-2.2 network, which is configured in a 3D toroidal topology.

The software environment on the Cray XMT includes a custom, multithreaded operating system for the Threadstorm, a parallelizing C/C++ cross-compiler targeting Threadstorm, a standard Linux 64-bit environment executing on the service and I/O nodes, and the necessary libraries to provide communication and interaction between the two parts of the XMT system. The parallelizing compiler generates multithreaded code that is mapped to the threaded capabilities of the processors automatically. Parallelism discovery happens fully- or semi-automatically by the addition of `pragmas` (directives) to the C/C++ source code. This discovery focuses on analyzing loop nests and mapping the loop's iterations in a data-parallel manner to threads.

To understand the lightweight synchronization features of the XMT, we review two aspects of the programming model: full-empty bits and generic functions. Each 8-byte word of memory has an associated full-empty bit enabling lightweight synchronization operations. The software (compiler and runtime) allows programs to manipulate the full-empty bits with generic functions are executed atomically within one instruction cycle.

Special versions of load and store instructions are emitted by the compiler automatically as part of parallelization and can be used directly by programmers:

- $readxx$: Returns the value of a variable without checking the full-empty bit.
- $readfe$: Returns the value of a variable when the variable is in a full state, and simultaneously sets the bit to be empty.
- $writeef$: Writes a value to a variable if the variable is in the empty state, and simultaneously sets the bit to be full.
- $writxef$: Writes a value to a variable if without checking the full-empty bit.
- int_fetch_add: Atomically adds an integer value to a variable.

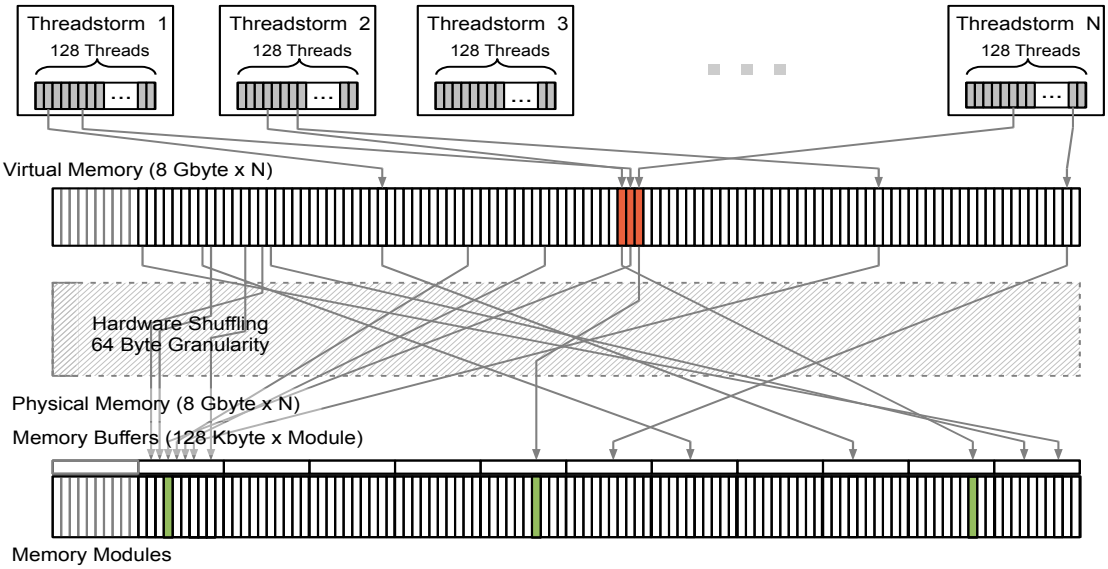

Figure 5: Cray XMT Threadstorm memory subsystem.

Much like the x86 which implements atomicity in the processor and memory controller, the XMT also implements atomicity in it's memory controller and processor, but differs in that the processor is able to context switch without the operating system, making context switch a side effect of memory semantics. This difference is critical to the XMT's programming model.

On XMT, the semantics of mutex locks are implemented by the hardware, which is only possible because the CPU can context switch without software intervention. Deadlocked software threads only use one stream's worth of state resources and no instruction issue resources regardless of how long the stream is stalled. Deadlocked loads are treated identically to data loads with long latency, simplifying the mental model of the system for debugging. Critical regions involving multiple locks can be implemented by nesting the emptying and refilling of memory locations.

The XMT provides tag bits for each 64-bit word of memory meaning that synchronization need not be rationed, and because the full/empty bit can be examined and modified as part of the data load or store already being performed, synchronization can casually be incorporated into algorithms. By extension, the XMT can be used to efficiently implement data flow algorithms.

3 Thread-local data

Privatization of variables in OpenMP is performed by adding the `private` clause to directives, listing variables from an outer scope which should be privatized in the parallel region. The loop induction variable is always considered local. The XMT compiler generally privatizes scalar variables and intermediate values automatically based on the dependence analysis, but also provides directives for explicit privatization. Both programming models honor ANSI-C's scoping rules with respect to automatic (stack) variables, meaning variables which are declared (and thus scoped) within a parallel region are privatized by virtue of already being on the thread's stack. An example of two variables, `tmp1` and `tmp2` declared with different scopes in ANSI-C but with the same effective scoping semantics due to modification with OpenMP pragmas as seen in figure 6.

3.1 Executing Once Per Thread

Hoisting of invariant operations out of loops to reduce or eliminate redundant operations is a basic loop optimization performed by compilers and programmers. Expensive operations such as system calls should be avoided inside perfor-

```
int i;
int tmp1;                          // Subroutine scoped automatic stack variable
#pragma omp parallel for private(tmp1)
for(i=0; i < npoints; i++) {
  int tmp2;                           // Thread Private Automatic stack variable
  tmp1 = ...;                       // tmp1 accesses are to thread-private copies
  tmp2 = ...;
  }
tmp1 = ...;                // Outer scoped tmp1 undefined after parallel region
```

Figure 6: Modifying the semantics of variable scoping to conform to parallel semantics.

mance critical loops, two common examples of idiom are performing I/O or allocating and freeing memory. These are frequently rewritten by hoisting the system calls out of the loop and issuing a single system call which does the work for all the iterations.

Consider the example in figure 7 which allocates temporary storage for each iteration of the loop. The compiler is not able to restructure this loop because the amount of storage needed is not known until runtime. However, the programmer may know `size` is never greater than `npoints`, meaning it would be safe to always allocate `npoints` of temporary storage. Some storage may go unused, but this space/performance tradeoff may be practical for a given problem and system size.

This idiom exposes a tension between classical scalar optimizations (hoisting of invariant expressions out of a loop) with the desire to push dependencies into inner loops to permit parallelization of the outer loop(s). In this instance, having the `malloc`/`free` pair inside the outer loop means there is no dependency on the temporary storage by ensuring that `tmp` is thread-private storage, however we wish to move the memory allocation call out of the loop, creating a new loop carried dependence on the storage for `tmp`.

A brute-force solution would pre-allocate all the memory needed by all the iterations before the loop begins, and the each iteration would index into it's private region in the large temporary storage space. This technique is inefficient when the number of iterations is larger than the number of threads because the storage needs to be replicated only to remove dependencies between threads, not between iterations. That is, a loop with 1 million iterations running on 4 processors needs temporary storage for 4 iterations, not 1 million iterations.

OpenMP syntax elegantly captures this idiom in the notion that parallel regions are separate from loops. Specifically, OpenMP programs can go parallel outside a loop, perform all the thread-private work before entering the loop, and then map the iterations of the parallel loop onto the threads which have already performed the thread-private work. This syntax is shown in figure 8.

By comparison the XMT's runtime does not even require that the number of threads working on a loop does not change while the loop is executing (in practice presently, this is constant for a single parallel region, but this is not required), making it challenging to express similar execution and storage semantics. In the revision 1.4 of the XMT's programming environment Cray has introduced a pragma creates a parallel region outside of a loop. The number of iterations is the number of streams the runtime acquires for that parallel region. This pragma is non-portable, has no OpenMP equivalent, may not work as expected when used in combination with other parallel regions, and will no longer work if/when the runtime is enhanced to permit threads to join and leave the parallel region during execution.

```
for(i=0; i < npoints; i++) {                    // Target loop to parallelize
  int size = ...data[i]...;
  float *tmp = malloc(sizeof(float) * size);      // Per-iteration allocation
  for(j=0; j < size; j++) {
    tmp[j] = ...;                                  // Thread-safe data updates
  }
  free(tmp);                              // Free temporary memory per iteration
}
```

Figure 7: Allocating and freeing temporary storage from inside a loop.

```
#pragma omp parallel
{
  float *tmp = malloc(sizeof(float) * npoints);    // Allocate once per thread
#pragma omp for
  for(i=0; i < npoints; i++) {
    size = ...data[i]...;
    for(j=0; j < size; j++) {
      tmp[j] = ....;                 // Thread safe data references and updates
    }
  }
  free(tmp);                              // Free temporary data once per thread
}
```

Figure 8: Efficient allocation and freeing of temporary storage inside a parallel region but outside of a loop.

4 Context Switching

The processor architecture of the Cray XMT is a *Barrel Processor* meaning that a single instruction execution pipeline is shared by many contexts which are selected from on every clock cycle. This is illustrated in figure 9. The term Barrel Processor comes from automatic machine gun designs in which a single barrel is served by a rotating cylinder or magazine containing ammunition.

By way of comparison, a x86 processor core has only a single hardware context and cannot change contexts on it's own. All context switch semantics on these systems are enforced by the operating system saving the processor's state, loading a new state, and resuming execution of a thread. This kind of time-multiplexed resource sharing has been implemented in time-sharing operating systems since the 1970's and can also be used to mimic parallel execution on serial hardware by over-subscribing the number of tasks to cores.

The efficiency of software managed context switching depends on the amount of state which needs to be saved (which can be several hundred bytes of state including all the multimedia, vector and floating point registers), how much work is performed between context switches, and the hardware's ability to store and re-load it's state. A typical x86 context switch can require several thousand cycles to store or load, requiring hundreds of thousands of instructions of work to amortize the cost of. For this reason, software context switch is not suitable for context switching at a fine grain such as cache misses.

Systems with efficient hardware for storing all state and bandwidth sufficient for fast switches can have more tasks mapped onto them without detrimental performance effects. For example,

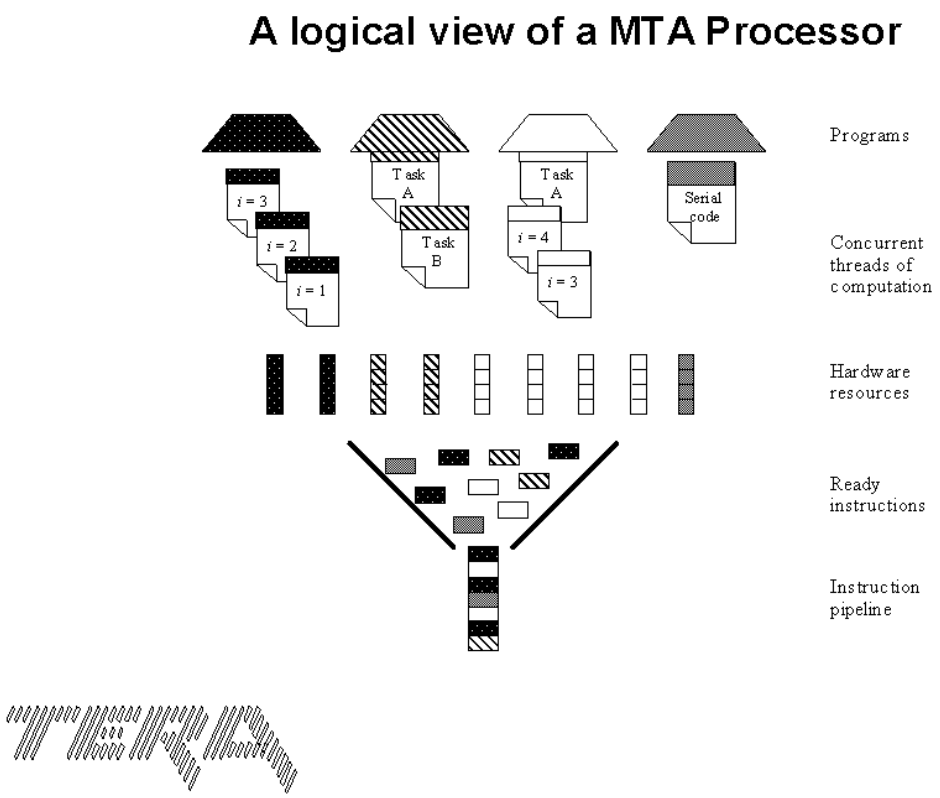

Figure 9: Logical diagram of instruction execution on the Cray XMT processor.

the T90's processors could context switch within a few instructions, and over-subscription (more than one software thread per hardware processor) of T90s was a standard use case. The XMT's processors does not need to store or re-load a thread's context during a context switch because the state is held entirely in banks of registers in the processor, thus a context switch can occur every clock cycle with no performance penalty.

Although most OpenMP programs assume there is one OpenMP thread per hardware processor, this is not required. The OpenMP programming model allows a program to run on any number of hardware resources equal to or less than the number of requested processors. This permits over-subscription (running more OpenMP threads than there are hardware processors) or for OpenMP to allocate a smaller number of threads than the number requested – or even no threads at all. Because of the relatively high cost of context switching on x86, it is uncommon to request more software threads than processors. However if this cost were reduced (or eliminated), it might possibly be advantageous to do so, and harmless to do even if there were no mechanical advantage to over-subscription.

5 Application Example

In this section we explore the different programming models by showing examples of implementations of similar tasks in the variety of programming models. We comment on the performance issues of each example.

5.1 Hashing

Hashing with chaining is a well-studied and well-established data structure and accompanying algorithms. The idea with chaining is that each *bucket* of the hash table is organized as a linked list. The strategy works best when the length of the linked lists are small.

Recently, a combination of "hashing with chaining" and "region based allocation" was presented [8]. This strategy, called HACHAR, is well-suited to a shared memory architecture. The general idea is to move synchronization points to the most localized position(s) possible. For example, we could lock the entire hashing structure, or provide locks for each bucket, or even move the synchronization out to the end of each chain, locking only when there is a need to extend the chain.

We note that a hashing repository may be used for a wide range of kernel applications, including: histogramming (counting the number of occurrences of each key), unique labeling (providing a globally unique label such as an integer value for each key), and existence lookup (checking to see if a key has been encountered before). While each of these variations has slightly different memory access patterns, they are nearly identical in the hashing aspect of their computation. We will use the histogramming variation in our example, and show only the `insert(Key key)` method. Note that at the end of processing, the hashing repository contains a collection of <`key, frequency`> pairs.

5.1.1 Hachar data structure

Recall that hashing with chaining involves hashing the key to get the bucket index followed by chaining forward in a linked list in search of the key. The type of processing is dependent upon the application variation (histogramming, unique id assignment, etc.). For the case of histogramming, we do the following:

- If the key is found, the associated frequency value is incremented.
- If key is not found, we insert a new key into the chain with an associated frequency value of one.

Although there is a good, general purpose memory manager on the XMT designed specifically for multithreaded architectures [9], our memory allocation pattern is well-known and specialized, so it makes sense to implement a region-based memory allocator to achieve better performance. The region-based memory allocator provides for all linked list nodes in the buckets to be allocated out of large regions (see Figure 10). Synchronization can occur at a global level (one lock for the entire structure), at an intermediate level (one lock for each bucket), or at a fine granularity than that (a lock at only the last node in each chain, allowing other threads to be visiting earlier nodes in the chain). Note that the fine-grain approach works because we only insert new nodes at the end of the chain.

The data structure is initialized by allocating a region to serve as the hash table, allocating a bucket sizes array, and writing zeroes in the bucket sizes array. In anticipating buckets that exceed a size of one, an empty region is allocated and linked from the hash table. To dismantle this data structure all that is needed is to free the bucket sizes integer array and each of the regions. If the data type of the key or value needs to be dismantled, then each of the array entries will need to be freed as well.

5.1.2 Insert into a HACHAR

A high-level description of the insert method is given in figure 11. Note that lines 2, 6 and 9 each require some type of protection against multiple thread updates. This can be done by allowing only one thread into this `insertHACHAR` method at a time or by locking parts of the data structure.

5.1.3 HACHAR on a Cray XMT

We first show the code (in figure 12) to perform the `insertEmpty` method on the XMT, which corresponds to line 2 of `insertHACHAR` in figure 11. This is followed by a sketch of the `while` loop

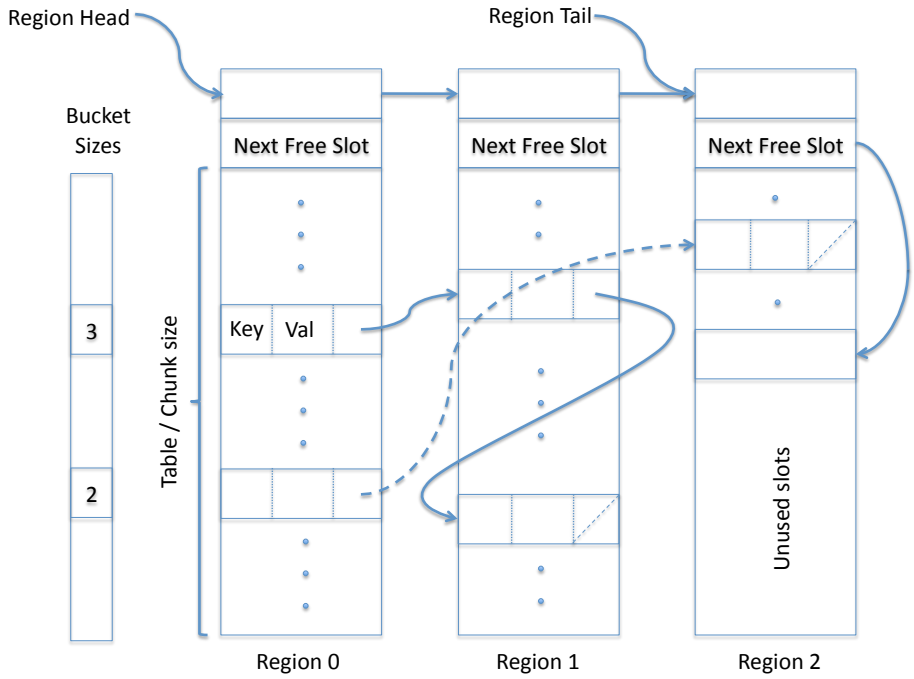

Figure 10: Data structure used to support the Hashing with CHaining And Region-based memory allocation(HACHAR).

```
   Procedure: InsertHACHAR(key)

 1: bucket = hashFunction(key)

    // try inserting into an empty bucket
 2: flag = insertEmpty(key, Value=1, bucket)

 3: while flag != success do
 4:     Walk down linked list looking for key                ▷ no locking used
 5:     if (key is found) then
 6:         increment value associated with key              ▷ synchronization required
 7:         flag = true
 8:     else
            // try to insert into the list
 9:         flag = growChain(key, Value=1)                   ▷ synchronization required
10:     end if
11: end while
```

Figure 11: Procedure to insert a key-frequency pair in a HACHAR structure. The increment operation at line 6 must be synchronized. Note that the `growChain` method may fail if some other thread increases the size of the chain between the time that we noticed that we had walked to the end of the list and the time that we tried to allocate a new node. If this occurs, we simply return to the top of the loop and continue walking the chain over the newer parts of the linked list.

(lines 3 through 11). Note that these XMT implementations assume a serial context but there may be many hundreds or even thousands of threads manipulating this HACHAR data structure concurrently.

The main while loop of the `insertHACHAR` function will iterate until the key has been processed (counted). Most of the operations can be done without the need for synchronization since the chains will grow only at the end of the linked list. That means that a search through an existing list can rely on unchanging data and structure.

If we run to the end of the chain, we must synchronize around the task of adding to the list. On the XMT we synchronize by setting the full-empty tag bit of the word holding the `next` link. When we have completed the update, we write a value to that link and clear the tag bit.

The details of the `growChain` function are not shown. This function uses a process similar to the `insertEmpty` to lock and then verify that the pointer is still `NULL` before linking in a new node. The new nodes are allocated from the pool of available link nodes in a region by using `int_fetch_add` on the `next_free_slot` index. Allocating a new region and linking it in also follows the same pattern of lock and then verify that the pointer is still `NULL`.

5.1.4 HACHAR in OpenMP

Since OpenMP uses a thread-centric synchronization programming model, we must lock certain parts of the code rather than locking the data to ensure data integrity across critical regions. To port the XMT code to an OpenMP machine, we must address synchronization with the `int_fetch_add` idiom and the `readfe`/`writeef` tag bit idiom.

The `int_fetch_add` idiom is the easier of the two and is addressed by simply providing an inline function that uses the OpenMP atomic pragma:

```
inline int
int_fetch_add(int * addr, int & value) {
   int old;
   #pragma omp atomic
   old = (*addr += value);
   return old-value;
}
```

This strategy will handle the increment found at line 6 of figure 11. The other idiom of the tag bits requires a little more attention. There are a couple of strategies for this: using an explicit lock by creating additional data for the lock and establishing a critical region of the code (essentially locking the code rather than the data). We show the second strategy in figure 14.

We can employ a similar technique to provide critical section locking in the `growChain` function. Note that this strategy may not scale very well because the critical section will not allow multiple threads into the section of code even though several of the threads needing to enter the critical section are trying to update different buckets of the hash table. The data-centric synchronization easily supports simultaneously updates to different buckets whereas the thread-centric synchronization either does not support simultaneous updates at all, or forces the use of an expensive software lock with separate data.

6 Conclusions and Future Work

Synchronization is a natural part of parallel programs which cannot be avoided. The Cray XMT makes synchronization primitives part of the memory model, as a result synchronization is easy to use because it is already part of the load-store model and it requires no special allocation or rationing. Communication-free parallelism (i.e.: vectorization) is a limited kind of parallelism and cannot be used on loops which require some kind of synchronization or output ordering. Synchronization is needed for implementing atomicity, enforcing order dependencies, and managing threads.

So called lock-free techniques that rely on atomic memory operations have locks implied the in mutual exclusion of the Atomic Memory Operation (AMO) instructions. Such instructions do not afford any concurrency and loops which operate on shared data using AMOs will

```
bool insertEmptyXMT(Key & key, Value & value, int bucket) {
  Flag flag = not successful;

  // try inserting into an empty bucket
  if (bucketSize[bucket] == 0) {
    int size=readfe(&bucketSize[bucket]);                              // lock
    if (bucketSize[bucket] == 0) {
      region0[bucket].key = key;
      region0[bucket].value = 1;
      region0[bucket].next = NULL;
      flag = successful;
      size = 1;
    }
    writeef(size);                                                     // unlock
  }
  return flag;
}
```

Figure 12: Implementation of the insertEmpty operation for the HACHAR strategy on the Cray XMT. This corresponds to line 2 of figure 11. Note that this function only *attempts* to insert into an empty bucket. If the bucket is not empty, this function will return `not successful`.

```
void insertHACHAR(Key & key) {
  unsigned int bucket = hashFunction(key);
  Flag flag = insertEmpty(key, Value=1, bucket);

  Node * ptr = & region0[bucket];
  while (flag != successful) {
    if (ptr->key == key) {
      // accumulate frequency count using atomic add
      int_fetch_add(& (ptr->Value), 1);
      flag = successful;
    } else if (ptr->next == NULL) {
      flag = growChain(key, Value=1);                          // this may fail
    } else {
      ptr = ptr->next;
    }
  }
}
```

Figure 13: Implementation of the `insertHACHAR` operation on the Cray XMT. Note that the call to `growChain` may fail if some other thread comes in and extends this chain between the time we notice the next link is `NULL` and the time inside of `growChain` where we will lock the next link (using `readfe`).

have an execution time that is a function of the number of iterations, not the number of iterations divided by the number of processors. Fine grained synchronization on data exposes concurrency not present in algorithms which manage concurrency through task management, making the algorithms more algorithmically efficient and faster executing.

```
bool insertEmptyOpenMP(Key & key, Value & value, int bucket) {
  Flag flag = not successful;

  // try inserting into an empty bucket
  if (bucketSize[bucket] == 0) {
    #pragma omp critical(insertEmptySlot)                          // lock code
    {
      if (bucketSize[bucket] == 0) {
        region0[bucket].key = key;
        region0[bucket].value = 1;
        region0[bucket].next = NULL;
        flag = successful;
        size = 1;
      }
    }                                                                // unlock
  }
  return flag;
}
```

Figure 14: Implementation of the insertEmpty operation for the HACHAR strategy using OpenMP. This corresponds to line 2 of figure 11. Note that this function only *attempts* to insert into an empty bucket. If the bucket is not empty, this function will return `not successful`.

In addition to extended memory semantics, the XMT implements efficient context switching in hardware, making it possible to tolerate the latency of memory operations by executing software threads whose memory operations have already completed. It is important to note that this mechanism which masks hardware latency also masks algorithmic latency when producers and consumers of shared data are far apart in time. This synergy between the programming model and the hardware simplifies programming by turning most processor under-utilization problems into a matter of finding more threads to execute.

Unlike special hardware features which may be implementation dependent, parallelism is performance portable. A program which is written to exploit vector registers of a given length may need to be completely redesigned to expose more concurrency when the hardware's vector length is increased. However, multithreaded software is agnostic with respect to how the parallelism is implemented, making the implementation irrelevant. Furthermore, the multithreaded programming model presents a single programming model from the desktop to the supercomputer in which quality of service is determined by the amount of concurrency in hardware, not optimizations in software.

Acknowledgments

This work was funded under the Center for Adaptive Supercomputing Software - Multithreaded Architectures (CASS-MT) at the Dept. of Energys Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

About the Authors

Jace A Mogill, Research Scientist, studies hybrid-microparallel computer architectures and parallel programming models. He is an applications analyst that eventually took Alan Kay's advice to learn to build his own hardware. He can be reached at: Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O. Box 999,

MSIN J4-30, Richland, WA, 99352, USA, Email: jace.mogill@pnl.gov.

David J. Haglin, senior research scientist, is interested in algorithm design, algorithm theory, parallel algorithms, graph algorithms and data mining. He can be reached at: Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O. Box 999, MSIN J4-30, Richland, WA, 99352, USA, Email: david.haglin@pnl.gov.

References

[1] Cray Research Inc., "Cray X-MP Series of Computer Systems," 1985, Sales/Marketing Literature.

[2] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2002, pp. 73–82.

[3] E. Petrank, M. Musuvathi, and B. Steesngaard, "Progress guarantee for parallel programs via bounded lock-freedom," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 144–154.

[4] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2004, pp. 35–46.

[5] Advanced MicroDevices, *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*, 3rd ed. AMD Inc., 2009.

[6] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 28–34.

[7] D. Chavarría-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer, "Early Experience with Out-of-Core Applications on the Cray XMT," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, April 2008, pp. 1–8.

[8] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarría-Miranda, J. Mogill, and J. Feo, "Hashing Strategies for the Cray XMT," in *Proceedings of the 24nd IEEE International Parallel and Distributed Processing Symposium*, April 2010.

[9] S. Kahan and P. Konecny, ""MAMA!": a memory allocator for multithreaded architectures," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 178–186.