Optimizing Loop-level Parallelism in Cray XMT^{TM} Applications

Michael Ringenburg and Sung-Eun Choi
Cray, Inc.

May 1, 2009

ABSTRACT: In this paper, we describe how to write efficient, parallel codes for the Cray XMT^{TM} system, a massively multithreaded shared memory computer. To achieve good performance on Cray XMT systems, programs must exploit not only coarse-grained parallelism at the algorithmic level, but also fine-grained parallelism at the loop level. While the Cray XMT compiler is capable of performing the loop-level optimizations required to expose fine-grained parallelism, it can often do a better job when additional information is provided by the programmer via pragmas and language constructs. These "hints" enable the compiler to perform sophisticated transformations that ultimately result in highly parallel codes. The Canal tool, part of the Cray Apprentice2^{TM} performance tool suite, can be used to guide the programmer through the process of tuning a program. When properly optimized, programs written for Cray XMT systems can achieve significant speed up on problems that have never been shown to attain speed up on conventional multiprocessor systems.

KEYWORDS: Cray XMT, Compilers, Multithreading, Parallelism, Cray Apprentice2, Canal.

1 Introduction

The Cray XMT^{TM} system is a massively multithreaded shared memory system purpose-built for parallel applications that require a large shared address space. Examples of such applications include graph analysis, database queries, and other problems where partitioning the data precludes any potential gain from adding more processors. Such applications do not typically run well on conventional distributed memory systems due to the irregular nature of memory access patterns. The Cray XMT architecture enables many threads to be running concurrently (up to 128 per processor) so that the memory accesses from individual threads can be overlapped with those of other threads, effectively hiding some or all of the actual memory latency.

Programs that exploit both coarse- and fine-grained parallelism are able to take advantage of this latency hiding capability and perform well on Cray XMT systems. Such programs draw on parallelism at the algorithmic level as well as at the loop and instruction level.

Programmers are typically better at identifying high-level parallelism at the algorithmic level. Such parallelism requires an insider's understanding of the problem space and objectives. In contrast, compilers are typically better at performing lower level optimizations such as loop parallelization. Often this requires the compiler to perform other transformations such as *loop collapse* and *scalar expansion*. Unfortunately, for languages such as C that were not original intended to be parallel, compilers have a difficult time making the conclusions required to perform such transformations and parallelize programmer-written loops. At the same time, programmers have a difficult time understanding what exactly the compiler needs to know in order to best optimize a loop. Both sides must work together to create an optimized program that runs well on the hardware.

In this paper, we describe how to write efficient programs for the Cray XMT system. The Cray XMT compiler is capable of performing sophisticated transformations that ultimately enable parallelization of complicated hand-written loops, providing certain conditions are guaranteed. Using the Canal tool, part of the Cray Apprentice2^{TM} tool suite, users can determine why loops were not parallelized and then use pragmas and other language constructs to guide the compiler to do the necessary transformations to enable parallelization.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the Cray XMT architecture and the programming environment, including the Canal tool. In Section 3, we detail specific examples of how the Cray XMT compiler identifies parallelism in user codes, and in Section 4 we describe how that parallelism is implemented. Finally, in Section 5 we walk through an example to illustrate how a programmer can use Canal to determine changes that enable the compiler to generate highly efficient code.

2 Overview of the Cray XMT System

A Cray XMT system is a distributed, global shared memory machine built to leverage Cray's MPP system design. A Cray XMT system includes a compute partition comprised of Cray Threadstorm^{TM} processors and a service partition comprised of AMD Opteron^{TM} processors, which can be configured for I/O, login, or system functions. The network topology is a 3-D torus, as in the Cray MPP systems. Both partitions are made up of blades of four processors each; the processors on a blade are all of one type, either Cray Threadstorm processors or AMD Opteron processors. The compute partition runs the Cray MTK^{TM} operating system, a single system image operating system based on BSD; the processors on the service partition utilize the Cray Linux Environment^{TM} (CLE).

A single Cray Threadstorm processor has 128 hardware contexts (or *streams*) and a 64 KB, 4-way associative instruction cache shared by all the streams. Each stream consists of 32 general purpose registers, eight target registers, and a *status word* that includes the PC. The processor will issue an instruction on every cycle in a round-robin fashion amongst the streams that are ready to execute an instruction; if no stream is ready to execute an instruction, the processor stalls. Up to three operations can be encoded in a single instruction word: one memory operation, one arithmetic operation, and one control flow operation.

The memory system is a global shared memory address space accessible by all Cray Threadstorm processors in the compute partition. Each processor can have up to 8GB of memory associated with it, all of which is accessible by every other processor in the system. Each 8-byte word of memory has 2 additional bits associated with it. One of the bits is called a *full-empty* bit, used to associate state with the word. A memory location state is considered *full* if the bit is set to one, *empty* if set to zero. The other bit is used by the runtime libraries to service and implement user traps (some of which are associated with the full-empty bit).

For more details on the XMT architecture, see Feo, *et al.* [2].

2.1 The Cray XMT Programming Environment and Tools

The Cray XMT Programming Environment includes a C and C++ compiler, a standard runtime library that includes support for multithreaded execution, a multithreaded debugger (mdb), the Cray Apprentice2^{TM} tool kit, as well as several auxiliary libraries such as a parallel random number generator (libprand), RPC library (libluc), and memory snapshot/restore facilities (libsnapshot). For the purposes of this paper, we will focus on the compiler and the Canal report of the Cray Apprentice2 suite.

The Cray XMT compiler supports C and C++ codes with extensions for parallelization and multithreaded execution. These XMT extensions include language extensions and compiler directives. In addition, the compiler can detect loop-level parallelism for multithreaded execution.

2.1.1 Language extensions

The Cray XMT compiler recognizes two type qualifiers, *sync* and *future*, that are used to indicate that the full-empty bits should be utilized whenever accessing the variable. Specifically, for sync qualified variables, a use (load) of the variable can only proceed if the variable's state is full; upon completion of the load, its state is set to empty. An assignment (store) to a sync variable can only proceed if the variable's state is empty; upon completion of the store, its state is set to full. Loads and stores that are issued when the variable is not in the required state will block until the variable's state changes. State changes occur atomically along with the operation.

Future variables are similar to sync variables, except that a load will only proceed if the state is full (and leaves the state full) and a store will also only proceed if the state is full (and leaves the state full). Future variables are typically used in *future* statements. Future statements define a block of work (referred to as a *future*) that is to be executed by some thread. A future variable can be associated with a future statement to guarantee that the future has been executed. Upon completion of the future, the future variable associated with the future statement is set to full. Any uses of the future variable will block until the future has been executed. Future statements describe what we refer to as *explicit parallelism*, because the parallelism is made explicit by the programmer.

2.1.2 Loop-level parallelism and Canal

In addition to future statements, the Cray XMT compiler supports *implicit parallelism* in the form of compiler-generated loop-level parallelism, the focus of this paper. If the compiler can determine that a loop can be safely executed in parallel, it will generate loop parallelism by distributing the iterations of a loop across multiple software threads. The runtime library assigns these threads to hardware streams, which then execute them in parallel. The compiler also inserts calls to create (fork) and terminate (join) threads.

The results of loop parallelization and other transformations can be seen in Canal. The Canal report has two parts—an annotated view of the compiled source code, and a report containing additional information about loops in the code. The annotated source view contains annotations for each loop in a user's code, as well as a selection of messages about the transformations that were applied to the code. For example, the following snippet tells us that two nested loops were parallelized, and that *reduction* and *manhattan loop collapse* (indicated by the `m`) were used (Section 3.2 discusses these transformations in detail):

```
        | for (int i=0; i<sz; i++) {
        |   int begin = index[i];
        |   int end = index[i+1];
        |   for (int j=begin; j<end; j++)
7 PP:m$ |     m[i]+=int_fetch_add(&a[j],1);
** reduction moved out of 1 loop
        | }
```

The other piece of the Canal report is the additional loop information section. This section contains details about loops, such as scheduling information and instruction counts. An example is shown in Figure 1.

Throughout the remainder of this paper, as we describe various optimizations, transformations, and conditions for parallelization, we will show how they can be viewed in the Canal report.

3 Identifying parallelism

The Cray XMT compiler attempts to parallelize user codes whenever it deems it safe and potentially profitable. Automatic dependence analysis is used to identify loops whose iterations are independent and can thus be executed concurrently. Users can insert pragmas to aid dependence analysis in the presence of aliasing. Loop transformations are also automatically applied to remove unnecessary dependencies.

In this section, we describe how the compiler identifies loops that are safe to parallelize. We start by discussing the conditions the compiler looks for to establish whether it is safe to parallelize a loop. We then discuss the transformations that the compiler uses to remove unnecessary dependencies, and finally talk about how user inserted pragmas can assist the compiler with this process.

3.1 Conditions for safe parallelization

Fundamentally, the Cray XMT compiler has two requirements in order to safely parallelize a loop. First, it must be able to prove that the loop's iterations can be safely executed in parallel. Second, the compiler must be able to figure out how to schedule the iterations prior to entering the loop. These requirements result in four conditions that the compiler looks for to determine whether a loop can be parallelized:

1. The number of loop iterations can be determined prior to the execution of the loop.

2. The loop does not contain any early exits, such as `return` or `break` statements.

3. There are no data dependencies between iterations of the loop.

4. The loop does not cause any observable side effects, such as I/O.

Condition 1 relates to the requirement that the compiler be able to figure out how to schedule the loop's iterations. Unless the compiler knows how many iterations will execute, it cannot schedule them—otherwise, it may schedule iteration $N + 1$ of a loop that should only iterate N times. For example, the compiler will not parallelize the loop:

```
Parallel region  37 in main
       Multiple processor implementation
       Requesting at least 60 streams

Loop  38 in main at line 107 in region 37
       Stage 1 of recurrence
       Loop summary: 1 memory operations, 0 floating point operations
                2 instructions, needs 45 streams for full utilization
                pipelined

Loop  39 in main at line 107 in region 37
       Stage 2 of recurrence
       Loop summary: 4 memory operations, 0 floating point operations
                4 instructions, needs 45 streams for full utilization
                pipelined
```

Figure 1: An example of the additional loop information presented in the Canal report. Here we see loops 38 and 39 are two stages of a recurrence computation (see Section 3.2.3), and are both part of parallel region 37 (see Section 4.1). We also see information about instruction counts and requested streams.

```
  for(int i=0; i < num_its; i++) {
    foo[i] = 2 * i;
    num_its -= bar[i];
  }
```

because the line which decrements `num_its` changes the loop upper bound during the loop's execution, making it impossible to determine the number of iterations prior to entering the loop. We can see this in the Canal report:

```
1 X |   for(int i=0; i < num_its; i++) {
1 X |     foo[i] = 2 * i;
1 X |     num_its -= bar[i];
    |   }
```

The `X` annotation indicates that the loop was not parallelized due to structural problems—in this case, an inability to compute the number of iterations.

Condition 2 also relates to the scheduling requirement. An early exit from the loop bypasses iterations that would otherwise have executed according to the compiler's calculation of the iteration count. For example, the compiler can parallelize the loop

```
for (int i = 0; i < n; i++)
  my_array[i] = sqrt(i);
```

because it knows that it will iterate `n` times. However, the compiler can not parallelize this loop:

```
for (int i = 0; i < n; i++) {
  my_array[i] = sqrt(i);
  if (rare()) break;
}
```

If `rare()` is true for an iteration, the later iterations should not execute. If the loop is parallelized, though, some of these iterations may execute prior to or in parallel with the iteration where `rare()` is true. The Canal report for this loop shows us that multiple exits prevented parallelization:

```
   1 X |   for (int i = 0; i < n; i++) {
** loop exit
** multiple exits
   1 X |     my_array[i] = sqrt(i);
   1 X |     if (rare()) break;
       |   }
```

Condition 3 means that the results of one iteration of the loop will not affect any of the other iterations. If this were not true, the reordering and interleaving of iterations that occurs when a loop is parallelized could result in incorrect behavior. For example, consider the loop in the following routine:

```
void foo(int *a, int *b, int n) {
  for (int i = 0; i < n; i++)
    a[i] = b[i];
}
```

If we call this routine with parameters a and b pointing to overlapping memory we will create a data dependence between iterations. Consider the call:

```
foo(&bar[0], &bar[1], 100);
```

In this call to foo(), the i=1 iteration of the loop will write to the memory location bar[1], which is also read by the i=0 iteration. If this loop is not parallelized, the i=0 iteration will complete before the i=1 iteration begins and the memory at bar[0] will get set to the original value of bar[1]. If, however, this loop were parallelized, the i=1 iteration might occur before the i=0 iteration. The memory at bar[0] would then end up with whatever value was written to bar[1] in the i=1 iteration. Because of this data dependence, the compiler will not parallelize the loop. This is shown in the Canal report:

```
    | void foo(int *a, int *b, int n) {
    |   for (int i = 0; i < n; i++)
1 S |     a[i] = b[i];
    | }
```

The S annotation indicates that the compiler did not parallelize the loop due to a data dependence associated with the source line where the S appears. If we know that a and b will never refer to overlapping memory we can get this loop to parallelize with the noalias pragma or the restrict type qualifier, as detailed in Section 3.3.

The final condition, number 4, simply ensures that any observable side effects will not occur out of order. For example, consider the following loop:

```
for(int i = 0; i < 1000; i++)
  printf{"Number %d\n", i);
```

This loop should count up from 0 to 999. However, if it is parallelized the output may appear out of order. For this reason, the compiler will avoid parallelizing loops with function calls that may generate side effects. Canal indicates this with the function with unknown side effects message:

```
        |   for(int i = 0; i < 1000; i++)
    1 S |     printf("Number %d\n", i);
**function with unknown side effects:printf
        |   }
```

If we do not care about possible reordering of side effects we can use pragmas to parallelize this loop, as detailed in Section 3.3.

3.2 Compiler transformations

In Section 3.1, we described the conditions that are necessary for the Cray XMT compiler to parallelize a loop. However, many loops that at first glance may not appear to meet these conditions can be transformed into equivalent loops that are parallelizable. As a simple example, loops that contain function calls with unknown side effects can sometimes be parallelized if the compiler inlines the call. The compiler aggressively seeks to identify and transform loops that would benefit from these optimizations.

In this section, we describe some of the most commonly applied automatic loop transforms used by the compiler, and show examples of loops that benefit. In particular, we discuss scalar expansion, reductions, linear recurrences, and various flavors of loop collapse.

3.2.1 Scalar Expansion

Scalar expansion is a commonly used loop transformation that eliminates data dependences due to unnecessary sharing of scalar variables. For example, consider the following loop:

```
for (i = 0; i < n; ++i) {
  t = sqrt(b[i]);
  ...
  a[i] = t + 5;
}
```

In this loop, every iteration will be writing to and reading from the same integer variable t. This creates a data dependence because each iteration must read the value it wrote to t before another iteration comes along and overwrites t.

However, the compiler is able to recognize that each iteration only needs the value that *it* wrote to t, and does not care about the values written by other iterations. Thus, the scalar integer t can be converted into an array of integers t[n], and each iteration can be rewritten to use a separate element of the array. The compiler will use this knowledge to automatically transform the above code into the following equivalent, but dependence-free, version:

```
for (i = 0; i < n; ++i) {
  t[i] = sqrt(b[i]);
  ...
  a[i] = t[i] + 5;
}
```

The compiler then successfully parallelizes the loop, as seen in the Canal report:

```
      | for (i = 0; i < n; ++i) {
5 P:e |   t = sqrt(b[i + 1]);
          ...
5 P   |   a[i] = t + 5;
      | }
```

The `P` annotation indicates that the loop was parallelized, and `e` indicates that the compiler automatically applied scalar expansion.

3.2.2 Reductions

Reduction is a loop transformation used to parallelize loops containing associative updates of scalar variables. The Cray XMT compiler supports sum, product, minimum, and maximum reductions. For example, consider the following loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    sum = sum +
          array[i*m + j];
    if (array[i*m + j] < min)
      min = array[i*m + j];
  }
}
```

This loop contains a dependence on the variable `sum` (and `min` as well), because interleaving of the reads and writes to `sum` from different iterations could result in incorrect computation of the final value. This can be solved by making the read and write behave as a single atomic memory operation (AMO), using either the Cray XMT system's `int_fetch_add` operation or its full/empty bits. This would allow the loop to be parallelized, but has the potential to cause *hotspotting*—a degradation of network performance due to saturation of network links. The AMO solution causes every thread to send memory references to the same memory address inside a tight loop, which can create hotspotting on the links leading to the node containing that address.

To avoid the potential hotspotting inherent in the AMO solution, the compiler instead converts loops containing associative updates into reductions. Each thread computes the operation over a subset of the input data and saves the value in a private variable. The threads then combine their results (taking advantage of the associative nature of the operation) in a tree like fashion to obtain the final result.

The Canal report displays reductions:

```
        |   for (int i = 0; i < n; i++) {
        |     for (int j = 0; j < m; j++) {
36 PP:$ |       sum = sum + array[i*m + j];
** reduction moved out of 2 loops
36 PP:$ |       if (array[i*m + j] < min)
** reduction moved out of 2 loops
        |         min = array[i*m + j];
        |     }
        |   }
```

The reduction message appears after the line containing the associative update that was replaced with a reduction. The message also indicates the number of nested loop levels out of which the compiler hoisted the global combining phase.

3.2.3 Linear Recurrences

The Cray XMT compiler is also able to parallelize certain loops that compute linear recurrences. Broadly, a recurrence computation computes each element of an array based on the results of computing prior elements, for example:

```
for (int i = st; i < size; i++)
  a[i] = foo(a[i-1],a[i-2],...,a[i-st]);
```

In the general case, loops of this form can not be parallelized due to the dependences on previous iterations. However, certain recurrences known as linear recurrences can be parallelized. In a linear recurrence, the computation of element i has the form

$$a[i] = a[i-k] * f[i] + g[i]$$

where k is a constant, and f and g can be replaced by scalars or constants. A common example is a loop that computes the *prefix sum* of an array:

```
for (int i = 1; i < size; i++)
  array[i] = array[i] + array[i - 1];
```

The Cray XMT compiler can parallelize linear recurrences where $k \leq 3$ using a technique based on cyclic reductions.

We illustrate this process with the prefix sum example from above (where $k = 1$). The compiler starts by breaking the array into contiguous blocks. Threads claim blocks and compute the recurrence over each block, pretending that all values before the block are 0. After a block is computed, we store the result of the final element:

```
id = block_number;
start = block_start[id];
end = block_start[id + 1];
previous_1 = 0;
for (int i = start; i < end; i++) {
  temp_array[i] = previous_1 + array[i];
  previous_1 = temp_array[i];
}
block_result_1[id] = previous_1;
```

Once all of the blocks have been computed, each block combines its final result with the final results of all of the previous blocks. This combination is done in parallel in a tree-like manner similar to reductions (as described in Section 3.2.2). This result is then used as a starting point to recompute the values of the next block (e.g., we use the combined result of the first three blocks to recompute the fourth block). In our prefix sum example, this step looks like:

```
id = block_number;
start = block_start[id];
end = block_start[id + 1];
previous_1 = combined_result_1[id - 1];
for (int i = start; i < end; i++) {
  array[i] = previous_1 + array[i];
  previous_1 = array[i];
}
```

The Canal report's annotated source listing shows users where the compiler applied the parallel linear recurrence transformation:

```
    | int recur(int *array, int size) {
    |   for (int i = 1; i < size; i++)
2 L |     array[i] = array[i] + array[i-1];
    |   return array[size-1];
    | }
```

The `L` annotation indicates that the compiler used cyclic reduction to parallelize a loop computing a linear recurrence.

3.2.4 Nested parallelism and loop collapse

The Cray XMT compiler has two approaches for handling nested parallel loops: nesting parallel regions (parallel regions are a blocks of code executed in parallel, and surrounded by a single fork and join—see Section 4.1 for more details), and loop collapse. Consider the following pair of parallel loops:

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    foo[i] += bar[j];
```

If the compiler parallelized these loops with nested parallel regions, the code would first go parallel for the outer loop. Each iteration would then go parallel again to execute the inner loop in parallel. Parallel region startup and teardown has a high overhead, however, so this approach is costly. In addition, multiple parallel regions can compete with each other for streams. Thus, the compiler's preferred approach is to collapse the loops into a single equivalent loop. In this case, that collapsed loop might look like:

```
for (int k = 0; k < n*m; k++) {
  i = k / m;
  j = k % m;
  foo[i] += bar[j];
}
```

The collapsed loop can then be parallelized with a single fork and join.

In the rest of this section we describe two of the most common forms of loop collapse—rectangular collapse and manhattan collapse. Rectangular collapse is used when the bounds of the inner loop are fixed across all iterations of the outer loop, while manhattan collapse is used when the bounds of the inner loop may vary between iterations of the outer loop.

Rectangular loop collapse The compiler uses rectangular loop collapse to collapse two perfectly nested parallel loops (i.e, the outer loop contains only the inner loop and nothing else) where the bounds of the inner loop are known to be fixed across all iterations of the outer loop. The loop nest in the following example is eligible because the inner loop is perfectly nested and has bounds `y_start` and `y_end`, neither of which is changed inside the outer loop:

```
for (x = x_start; x < x_end; x++)
  for (y = y_start; y < y_end; y++)
    grid[x][y] = 1;
```

If the two loops are not perfectly nested the compiler can sometimes distribute (i.e, split) the outer loop into multiple loops such that one of these loops contains only the perfectly nested inner loop. For example, consider the following loop:

```
for (x = x_start; x < x_end; x++) {
  row_start[x] = y_start;
  for (y = y_start; y < y_end; y++)
    grid[x][y] = 1;
}
```

The compiler can split this loop into two loops:

```
for (x = x_start; x < x_end; x++)
  row_start[x] = y_start;

for (x = x_start; x < x_end; x++)
  for (y = y_start; y < y_end; y++)
    grid[x][y] = 1;
```

The inner loop is now perfectly nested, and the compiler can collapse it with the outer loop. As we will see in Section 4.1, these loops can still be implemented as part of a single parallel region.

In a rectangular collapse, the nested pair is transformed into a single loop whose iterations simulate every iteration of the original nested pair. The first step is to normalize the loops to start at 0:

```
for (i = 0; i < x_end - x_start; i++)
  for (j = 0; j < y_end - y_start; j++) {
    x = i + x_start;
    y = j + y_start;
    grid[x][y] = 1;
  }
```

The next step is to replace the loop nest with a single loop whose upper bound is the product of the upper bounds of the previous pair:

```
int new_bound = (x_end - x_start) *
                  (y_end - y_start);
for (k = 0; k < new_bound; k++) { ... }
```

Finally, the compiler rewrites the body of the loop to compute the original index values:

```
int new_bound = (x_end - x_start) *
                  (y_end - y_start);
for (k = 0; k < new_bound; k++) {
  i = k / (y_end - y_start);
  j = k % (y_end - y_start);
  x = i + x_start;
  y = j + y_start;
  grid[x][y] = 1;
}
```

Manhattan loop collapse The rectangular collapse described above works whenever the inner loop bounds are the same for every iteration of the outer loop. Many of the graph-based codes that are frequently run on the Cray XMT system, however, contain loop nests that do not meet this condition. For example:

```
for (int i=0; i<sz; i++) {
  int begin = index[i];
  int end = index[i+1];
  for (int j=begin; j<end; j++) {
    total += adj[j];
  }
}
```

In this situation, the compiler will instead attempt to use a manhattan loop collapse. A loop nest is eligible for manhattan collapse if both loops are parallelizable, and the inner loop is perfectly nested except for statements that compute the inner loop bounds. If the nesting is not perfect, the compiler will attempt to make it perfect by using loop distribution (as described previously).

A manhattan loop collapse has two steps. First, the compiler creates a new loop that computes the number of inner loop iterations. The result of this loop is an array `tripcnt[]` whose i-th element contains the number of inner loop iterations executed during the first i iterations of the outer loop. Thus, `tripcnt[0]` is 0, `tripcnt[1]` is the number of inner loop iterations during the first outer loop iteration, `tripcnt[2]` is the number of inner loop iterations during the first two outer loop iterations, etc. In our example, the `tripcnt[]` loop looks like:

```
tripcnt[0] = 0;
for (int t = 0; t < sz; t++) {
  int begin = index[t];
  int end = index[t+1];
  int iterations = end - begin;
  tripcnt[t+1] = tripcnt[t] + iterations;
}
```

The compiler recognizes that this loop is just a prefix-sum linear recurrence calculation, and parallelizes it accordingly (see Section 3.2.3).

The second step of the manhattan collapse creates a single loop which simulates the execution of the original loop nest. The upper bound of the new loop will be `tripcnt[`N`]`, where N is the number of iterations of the outer loop. In our example, this would be:

```
for (int k = 0; k < tripcnt[sz]; k++) {...}
```

Here, iteration `k` will correspond to the `k`-th iteration of the original inner loop. Inside the new loop, we use the `tripcnt[]` array to determine which values of the inner and outer loop indices correspond to the `k`-th iteration of the collapsed loop. The outer loop index will be the element of `tripcnt[]` containing the highest element less than or equal to `k`. This is computed by a routine which performs a binary search on the `tripcnt[]` array (note that `tripcnt[]` will be an ordered array):

```
  int i = highest_less_than(tripcnt, k);
```

We determine the inner loop index by computing the difference between `k` and `tripcnt[i]`:

```
  int j = k - tripcnt[i];
```

We then execute the inner loop body, using the computed indices. Putting it all together, we have:

```
tripcnt[0] = 0;
for (int t = 0; t < sz; t++) {
  int begin = index[t];
  int end = index[t+1];
  int iterations = end - begin;
  tripcnt[t+1] = tripcnt[t] + iterations;
}
for (int k = 0; k < tripcnt[sz]; k++) {
  int i = highest_less_than(tripcnt, k);
  int j = k - tripcnt[i];
  total += adj[j];
}
```

The `m` annotation in the Canal report's annotated source listing indicates that the compiler performed a manhattan loop collapse:

```
        | for (int i=0; i<sz; i++) {
 6 P:e  |   int begin = index[i];
 6 P:e  |   int end = index[i+1];
        |   for (int j=begin; j<end; j++) {
10 PP:m$ |     total += adj[j];
** reduction moved out of 2 loops
        |   }
        | }
```

In this loop, we can also see that the compiler applied the reduction and scalar expansion transformations.

3.3 Pragmas

The Cray XMT compiler supports a set of pragmas that programmers can use to provide more information about their codes, and that can thus help identify parallelism. These pragmas have the form:

```
#pragma mta assert ...
```

The `mta assert` pragmas can be thought of as "promises" to the compiler about properties of user codes. These promises can be a very powerful way to improve parallelization and thus performance of applications. However, it is important to note that the compiler will rely on the accuracy of these promises, so it is critical to ensure that the promises are in fact true. Otherwise, the compiler may make incorrect assumptions and generate bad code. In this section, we discuss three of the most commonly used assert pragmas: `assert noalias`, `assert no dependence`, and `assert parallel`. Additional pragmas are described in the Cray XMT Programming Environment User's Guide[1].

The `assert noalias` pragma takes a list of pointer or array variables as an argument, and is a promise to the compiler that every variable `x` on the list points to memory which is only accessed through `x`. It applies to the entire scope of `x`. For example, consider the following code:

```
void vector_x2(int *x, int key,
               int n) {
  int *z = fetch_vector(key, n);
  for (int i = 0; i < n; i++) {
    x[i] = 2 * z[i];
  }
}
```

The compiler will not parallelize this loop, because it does not know if the write to `x` modifies memory that is read in another iteration through `z`. We can see this in the canal report:

```
    | void vector_x2(int *x, int key,
    |                int n) {
    |   int *z = fetch_vector(key, n);
    |   for (int i = 0; i < n; i++) {
1 S |     x[i] = 2 * z[i];
    |   }
    | }
```

However, we can get this loop to parallelize by adding an `assert noalias` pragma:

```
 void vector_x2(int *x, int key, int n) {
#pragma mta assert noalias *x
   int *z = fetch_vector(key, n);
   for (int i = 0; i < n; i++)
     x[i] = 2 * z[i];
 }
```

In this code, the programmer asserts that `x` does not refer to the same memory as `z` (or any other variable in its scope). The compiler uses this information to deduce that there are no dependencies between iterations and that the loop is thus parallelizable.

The C99 `restrict` type qualifier is also available in the Cray XMT compiler. Pointers with the `restrict` qualifier are treated like `noalias` variables, so the previous example can be rewritten as follows and still parallelize:

```
void vector_x2(int * restrict x, int key,
               int n) {
  int *z = fetch_vector(key, n);
  for (int i = 0; i < n; i++)
    x[i] = 2 * z[i];
}
```

The `restrict` qualifier can be used in some situations where `noalias` is not allowed, such as in type declarations (e.g., fields of struct declarations can have restrict-qualified pointer types). On the other hand, programs using `restrict` may be less portable because some C++ compilers do not support the `restrict` keyword (unsupported keywords lead to syntax errors, whereas unsupported pragmas are typically ignored).

The `assert no dependence` pragma is a promise to the compiler regarding dependencies between iterations of a loop. It must appear directly before the loop to which it refers. There are two forms of the `no dependence` pragma—one with a comma-separated list of noalias (or restrict) variables, and the other without a variable list.

The form that takes a variable list promises that references to memory through variables on the list do not create any dependencies between iterations. For example, consider the following loop:

```
for (int i = 0; i < num_move; i++)
  arr[dst[i]] = arr[src[i]];
```

The compiler will not parallelize this loop because there may be overlap between the elements of `arr` pointed to be `src` and `dst`. This overlap would create a dependence. If, however, the programmer knows that `src` and `dst` will point to disjoint subsets of `arr`, they can use a `no dependence` pragma:

```
#pragma mta assert no dependence *arr
for (int i = 0; i < num_move; i++)
  arr[dst[i]] = arr[src[i]];
```

This loop will parallelize.

The alternate form of `assert no dependence` takes no variable list, and is a promise that there are *no* data dependencies between iterations. The no variable list form is useful when dependencies involve variables that can't be declared as `noalias` or `restrict`. For example, consider the following code:

```
int *dst = &arr[new_start];
for (int i = 0; i < num_move; i++)
  *(dst++) = arr[src[i]];
```

In this case, we can not declare `arr` to be `noalias` because `dst` does in fact refer to part of the array `arr`. This prevents us from using the variable list version of `assert no dependence`. We can still use the no-list version, however, and get parallelization:

```
#pragma mta assert no dependence
for (int i = 0; i < num_move; i++)
  *(dst++) = arr[src[i]];
```

The `assert parallel` pragma can be thought of as the "big hammer" of the Cray XMT compiler's assertion pragmas. It is a promise that the loop which follows the pragma can safely be executed in parallel *as written*. This does not guarantee that the loop will be parallelized (e.g., if the compiler can not determine an iteration count), but it is a strong suggestion. The `assert parallel` pragma will even cause side effects to be ignored. For example, this loop will parallelize despite the possibility that the output may appear out of order:

```
#pragma mta assert parallel
for(int i = 0; i < 1000; i++)
  printf("Number %d\n", i);
```

The `assert parallel` pragma can be a powerful tool to get difficult loops to parallelize; however, it should only be used as a last resort when other techniques fail. Use of `assert parallel` may limit optimizations and transformations that the compiler might otherwise perform, because the programmer is only telling the compiler that the loop is safe to parallelize, without explaining why. Since the compiler does not understand why it is safe to parallelize, it has to be careful not to do anything which might affect that. This can in turn hurt performance.

4 Implementing parallelism

Once parallel loops have been identified, the next step is to determine the most efficient implementation strategy. The Cray XMT compiler supports three styles of parallelism—single processor, multiple processor, and a highly dynamic style known as *loop futures*. The compiler also supports a number of different iteration scheduling strategies, and attempts to combine nearby parallel loops into parallel regions with a single fork and join to reduce overhead.

In this section we discuss these strategies, and show how you can view the implementation choices made by the compiler. We also discuss how you can override these decisions with pragmas.

4.1 Parallel regions

The Cray XMT compiler will attempt to merge nearby parallel loops, and possibly intervening serial code, into a single *parallel region*—a section of code executed by multiple threads and surrounded by a single fork and join to begin and end the parallelism. Barriers are added to ensure correct execution. This merging allows the compiler to avoid paying the overhead of any extra forks and joins.

A parallel region consists of a series of parallel loops, serial code sections, and barriers. Barriers are used to ensure that we complete sections of the region before the results are needed by subsequent code in the region. Whenever a thread hits a barrier, it waits there until every other thread reaches the barrier. For example, consider the following code structure:

```
LOOP A
STATEMENT B // depends on A
STATEMENT C // depends on A
LOOP D // depends on B,C
LOOP E // depends on B,C
```

The compiler can implement this as a single parallel region:

```
FORK
  LOOP A
  barrier()
  if (thread_id == 0)
    STATEMENT B
  else if (thread_id == 1)
    STATEMENT C
  barrier()
  LOOP D
  LOOP E
JOIN
```

Every thread will execute its portion of the iterations of LOOP A, then wait at the barrier until all of the other threads have completed their portions. Threads 0 and 1 will then execute STATEMENTS B and C in parallel and wait at the barrier. Finally, every thread will execute its portion of loops D and E. Since loop E does not depend on loop D, the threads will not wait between finishing their portion of D and beginning their portion of E.

The extent of parallel regions can be seen in the additional loop info section of the Canal report. Every loop reports either its region or the loop in which it is nested. If a loop is nested in another loop, users can refer to the outer loop to discover the region. For example, in the following Canal report loops 2 and 3 are part of parallel region 1:

```
Parallel region   1 in foo
...
Loop   2 in foo in region 1
...
Loop   3 in foo at line 7 in loop 2
```

4.2 Styles of parallelism

The Cray XMT compiler supports three styles of parallelism, each with differing overheads and performance characteristics. Single processor parallelism has the lowest overhead, but only takes advantage of the streams on a single processor of the Cray XMT system. Multiprocessor parallelism takes advantage of the streams on all of the processors allocated to the job, but has significantly higher overhead. Finally, loop future parallelism has the highest overhead, but is also the only form of parallelism that can dynamically grow to use more streams as they become available and/or are needed. The compiler will choose a parallelism style by default, but this choice can be overridden by user directives.

The parallelism style applies to an entire parallel region, and the overhead is paid once per region. Because of this, it rarely makes sense to specify different parallelism styles for nearby loops. It will force the loops into separate regions and likely cause additional overhead. For example, consider the following pair of loops:

```
// Small loop
for (int i = 0; i < SMALL_N; i++) { ... }
// Big loop
for (int j = 0; j < BIG_N; j++) { ... }
```

The user might be tempted to request single processor parallelism for the small loop here, thinking that the performance benefit of using multiple processors does not justify the higher overhead of multiprocessor parallelism. However, if we were to then request multiprocessor parallelism for the big loop (because there is much more to be gained from using many processors in that case), we would end up paying the overhead of a single processor fork and join, followed by the overhead of a multiple processor fork and join. On the other hand, if both are implemented using multiprocessor parallelism, the compiler can merge them into a single region and only pay the overhead of a single fork and join.

Single processor Single processor parallelism is typically used for small loops (loops that have a low iteration count and only do a small amount of work). The performance gains are limited because the loop can only take advantage of the streams available on a single processor. However, this form of parallelism is significantly cheaper than the others, and so can be beneficial when the amount of work to be parallelized is small. For example, the compiler will usually select single processor parallelism for this loop:

```
for (int i = 0; i < 500; i++)
  a[i] = b[i];
```

This can be seen in the Canal report by looking at the information for the region containing the loop:

```
Parallel region   1 in foo
       Single processor implementation
```

If the compiler can not estimate the amount of work done by a loop (e.g., if the iteration count is not a compile-time constant), it will assume that it is large and choose a different style of parallelism. The user can override this choice, however, via a command line flag or a pragma. The `-par1` flag causes the compiler to use single processor parallelism as its default for all loops in the program. Similarly, the pragma

```
#pragma mta parallel single processor
```

tells the compiler to use single processor parallelism for all subsequent loops in the source file in which it appears. It takes precedence over any conflicting command line flags. Finally, the pragma

```
#pragma mta loop single processor
  for (...) { ... }
```

tells the compiler to use single processor parallelism for the following loop, and takes precedence over any other flags or pragmas.

Multiprocessor Multiprocessor parallelism has higher startup overhead than single processor parallelism, but also allows your program to take advantage of streams on all processors allocated to your job. The compiler chooses it when the amount of work is large (or not known). For example, if we double the iteration count in our previous example, the compiler will choose multiprocessor parallelism:

```
for (int i = 0; i < 1000; i++)
  a[i] = b[i];
```

We can once again see this by viewing the additional information for the region containing the loop:

```
Parallel region   1 in foo
       Multiple processor implementation
```

The user can specify multiprocessor parallelism using the `-par` command-line flag (this is the default in the current version of the compiler), or via the

```
#pragma mta parallel multiprocessor
```

or

```
#pragma mta loop multiprocessor
  for (...) { ... }
```

pragmas. These flags and pragmas have the same meanings as their single processor equivalents.

Loop futures Loop future parallelism is a highly dynamic form parallelism that is useful for recursive parallel loops where the workload may vary greatly between iterations, such as in the following example from a recursive depth-first search routine called DFS:

```
#pragma mta loop future
for (i = firstNode; i < lastNode; i++) {
  int nbr = Neighbors[i];
  // Ensure that only one thread visits nbr
  int v = int_fetch_add(&Visited[nbr], 1);
  if (v == 0) DFS(nbr, A);
}
```

Unlike the other forms of parallelism, loop future regions are not allocated a fixed number of streams when the region starts up. Instead, the compiler produces pointers to a piece of code that executes iterations of the loop. These pointers, or *continuations*, are placed on a queue managed by the runtime library. As streams become available, they check the queue, grab any available continuations, and begin executing iterations of the loop. This allows the region to grow as resources are needed and become available, but adds significant overhead. The continuations from all loop future regions are placed into the same queue, so this can also help with load balancing across multiple levels of a recursive parallel loop (such as in the DFS example above). As with the other forms of parallelism, we can view loop futures in the Canal report:

```
Parallel region   1 in foo
       Implemented with futures
```

The compiler does not typically choose loop future parallelism because of the high overhead and difficulty of analyzing recursive loops. However, you can request it with the -parfuture flag or with the

```
#pragma mta parallel future
```

or

```
#pragma mta loop future
  for (...) { ... }
```

pragmas. These flags and pragmas have the same meanings as their single processor equivalents.

4.3 Loop scheduling

The Cray XMT compiler has a number of different idioms at its disposal for scheduling the iterations of a parallel loop. Each of these idioms provides a different set of trade-offs between overhead and load balancing. The compiler attempts to choose an idiom based on its analysis of the characteristics of the loop, but pragmas are once again available to allow the programmer to direct the compiler's decisions.

The cheapest of the scheduling idioms is *block scheduling*. In a block scheduled loop, each thread will execute a contiguous block of `iterations/threads` iterations. For example, if we have three threads executing a block scheduled parallel loop with 100 iterations, one thread will execute iterations 0-32, another iterations 32-65, and the third iterations 66-99. Each thread can quickly calculate the iterations it should execute at the beginning of the loop with no inter-thread communication, making this the cheapest scheduling technique. It may also allow for some reuse of register data since threads will be executing contiguous iterations. However, if the work done by each iteration varies significantly, some threads may end up doing more work than others. This results in poor load balancing, where some threads complete their work early and then have to wait for other threads with more work to finish.

Interleaved scheduling is another cheap scheduling idiom that can in some cases provide improved load balancing. In an interleaved parallel loop, each thread executes iterations separated by the number of threads. For instance, if there were three threads and 12 iterations, the first thread would execute iterations 0, 3, 6, and 9, the second iterations 1, 4, 7, and 10, and the third iterations 2, 5, 8, and 11. Like block scheduling, interleaved scheduling requires only cheap and local computations. In certain cases, interleaved scheduling can also provide better load balancing, e.g., with triangular loop nests:

```
for(int i = 0; i < n; i++)
  for (int j = 0; j < i; j ++)
```

In this example, later iterations of the outer loop will do significantly more work than earlier iterations. Interleaved scheduling will help load balancing because every thread will execute a similar mix of early and late iterations. On the other hand, interleaved scheduling may not allow for the same reuse of register data that is possible with block scheduling because threads do not execute contiguous iterations.

Dynamic scheduling provides good load balancing for more generalized cases of loops whose work varies between iterations. In a dynamically scheduled parallel loop, a shared counter is used to keep track of iterations. Each thread retrieves the value of the counter to determine which iteration to execute, and increments it so that the next thread will claim the next iteration. As threads complete each iteration, they once again fetch and increment the counter to claim another iteration. The compiler uses the Cray XMT system's atomic fetch-and-add instruction (`int_fetch_add`) to retrieve and update the counter in order to avoid races between threads. Dynamic scheduling provides good load balancing for many loops because threads never stop claiming and executing iterations until every iteration has been claimed by some thread—thus, we keep threads working as long as possible. However, dynamic scheduling has a high overhead because every thread must access the shared counter after every iteration. This adds an extra global memory reference per iteration, and may cause contention.

Block dynamic scheduling provides a compromise between the load balancing of dynamic scheduling and the low overhead of block scheduling. In a block dynamic scheduled loop, threads claim blocks of iterations instead of single iterations. This reduces the overhead by decreasing the number of accesses to the shared counter, while still maintaining some of the load balancing properties of dynamic scheduling.

Loop scheduling choices can be viewed in the Canal report by looking at the additional loop info section, e.g.:

```
Loop  34 in count2 at line 107 in region 33
       Interleave scheduled
```

Users can direct the compiler to use different scheduling idioms with the following pragmas:

```
#pragma mta block schedule
#pragma mta interleave schedule
#pragma mta dynamic schedule
#pragma mta block dynamic schedule
```

These pragmas apply only to the next loop.

5 An example loop

In this section we show how to apply the information in this paper and in the Canal report to improve the performance of an example code. The code we use contains a search-breakout style loop, where we iterate over an array until a sought element is found. This example initially does not parallelize, but we show how knowledge of the conditions necessary for parallelism (Section 3.1) combined with the information contained in the Canal report can be used to rewrite the code and achieve parallelism.

Consider the following routine that copies the elements from an array `b` into an array `a` until an element `key` is found in `b`:

```
bool foo(int *a, int *b, int n, int key) {
  int i;
  for (i = 0; i < n; i++) {
    if (b[i] == key) break;
    a[i] = b[i];
  }
  return (i < n);
}
```

If we compile this routine and view the Canal report, we will see that the for loop is not parallelized:

```
    1 X |   for (i = 0; i < n; i++) {
** loop exit
** multiple exits
    1 X |     if (b[i] == key) break;
** loop exit
    1 X |     a[i] = b[i];
        |   }
```

The Canal report tells us that this loop could not be parallelized because it has an early exit (the `break`) and thus violates condition 2 from Section 3.1. To fix this, we need to find a way to rewrite the loop and remove the `break` statement. We can do this by splitting the original loop into two loops. The first loop will identify the minimum index of `b` at which `key` occurs (this can be parallelized as a reduction, as described in Section 3.2.2). The second loop will then copy all the elements of `b` up to the index identified in the first loop:

```
bool foo(int *a, int *b, int n, int key) {
  int i;
  int found_index = n;
  for(i = 0; i < n; i++) {
    if (b[i] == key)
      if (i < found_index)
        found_index = i;
  }
  for(int i = 0; i < found_index; i++)
    a[i] = b[i];
  return (found_index < n);
}
```

If we compile this and view the Canal report, we see that the search loop now parallelizes as a reduction:

```
     | for(i = 0; i < n; i++) {
3 P:$|   if (b[i] == key)
** reduction moved out of 1 loop
     |     if (i < found_index)
     |       found_index = i;
     | }
     | for(int i = 0; i < found_index; i++)
4 S  |   a[i] = b[i];
```

However, the copy loop is not parallelized. The S annotation tells us that there is a dependence involving the line that copies from b into a. If we look more closely at this routine, we see that a and b are pointers that could point anywhere—including to overlapping memory, which would create a dependence. If we know that this routine will only be called with parameters that point to disjoint arrays, we can use the noalias pragma on a:

```
bool foo(int *a, int *b, int n, int key) {
#pragma mta assert noalias *a
  int i;
  int found_index = n;
  for(i = 0; i < n; i++) {
    if (b[i] == key)
      if (i < found_index)
        found_index = i;
  }
  for(int i = 0; i < found_index; i++)
    a[i] = b[i];
  return (found_index < n);
}
```

If we compile this new version, both loops parallelize:

```
     |#pragma mta assert noalias *a
     | int i;
     | int found_index = n;
     | for(i = 0; i < n; i++) {
3 P:$|   if (b[i] == key)
** reduction moved out of 1 loop
     |     if (i < found_index)
     |       found_index = i;
     | }
     | for(int i = 0; i < found_index; i++)
5 P  |   a[i] = b[i];
```

6 Summary

The Cray XMT compiler has sophisticated program analysis and transformation capabilities to help users take advantage of the massive multithreading possible on the Cray XMT system. In order to take full advantage of these capabilities, a user needs to have an understanding of how the compiler transforms and parallelizes codes. In this paper, we discussed these processes, and showed how to use the Canal report to gain further insights. Armed with this knowledge, and the feedback from the Canal report, programmers can tune their applications to achieve efficient, scalable loop parallelism.

Acknowledgments

The authors would like to thank the Cray XMT development and management team for their support of this work. We would also like to thank Gail Alverson, Steve Deitz, Greg Titus, and Karlon West for their helpful feedback on earlier drafts of this paper.

About the Authors

Michael Ringenburg and Sung-Eun Choi are engineers in the XMT group at Cray, Inc. They can be reached at Cray, Inc., 901 Fifth Avenue, Suite 1000, Seattle, WA 98164, (206)701-2000.

References

[1] *The Cray XMT^{TM} Programming Enviroment User's Guide.* March 2009.

[2] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.