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

Michael Ringenburg and Sung-Eun Choi

CUG 2009

Outline

- **Overview of the Cray XMT^{TM} system**
- Introduction to loop parallelism
  - Why do we care about loop parallelism?
  - Parallel execution of loops: threads and iterations
- Identifying parallelism
  - Conditions necessary for parallelism
  - Compiler transformations to augment parallelism
  - Pragmas to assist parallelization
- Implementing parallelism
  - Parallel regions
  - Single processor, multiprocessor, and loop future parallelism
- A parallelization example

Overview of the Cray **XMT™ system**

- The Cray XMT™ system is built on the Cray XT™ infrastructure
  - Uses the same cabinets, boards, scalable interconnect, I/O and storage infrastructure, user environment, and administrative tools … just changes the processor
- Architected for large-scale data analysis
- Exploits thousands of parallel threads accessing large irregular datasets
  - Hardware supports 128 concurrent threads per processor; runtime software supports “oversubscription”
  - Architecture supports scaling to over 8000 sockets and 1M threads
  - Architecture supports scaling to 128 terabytes of shared memory

Outline

- Overview of the Cray XMT™ system
- **Introduction to loop parallelism**
  - **Why do we care about loop parallelism?**
  - **Parallel execution of loops: threads and iterations**
- Identifying parallelism
  - Conditions necessary for parallelism
  - Compiler transformations to augment parallelism
  - Pragmas to assist parallelization
- Implementing parallelism
  - Parallel regions
  - Single processor, multiprocessor, and loop future parallelism
- A parallelization example

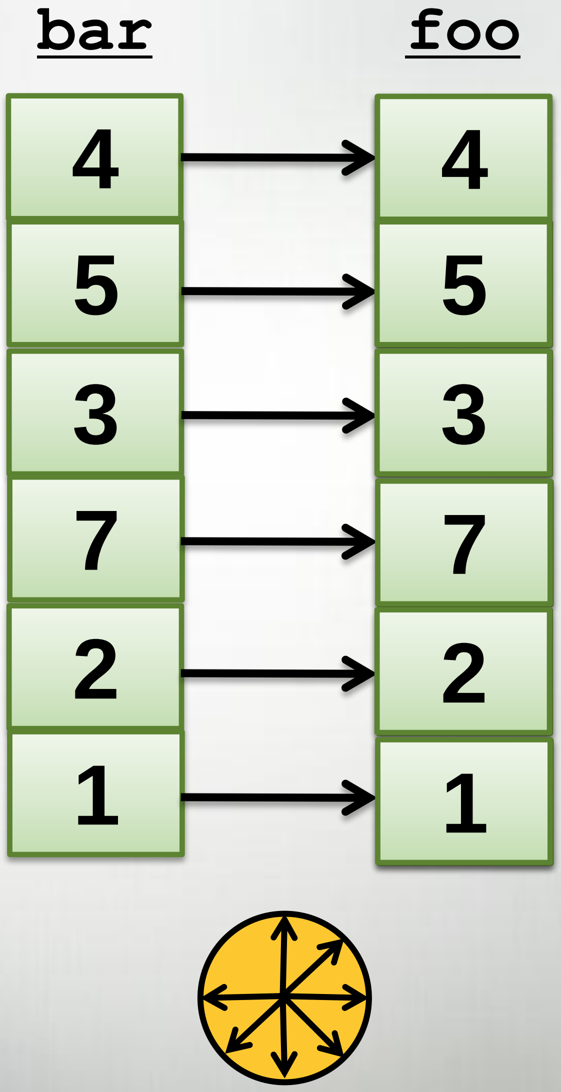
Why do we care about loop parallelism?

- Applications on the Cray XMT™ system require lots of parallelism to perform well.
  - Each processor has 128 hardware threads
  - A machine typically has hundreds of processors
  - If your application does not take advantage of these resources, it will not perform up to the capabilities of the machine.
- There are two main sources of parallelism in user applications
  - User-specified future-based parallelism
    - User specifies code that can run on another thread via a future statement
  - Compiler-generated loop parallelism (**focus of this talk**)
    - The compiler breaks up the loop iterations and runs them on different threads.
    - User may assist the compiler in this process

What is loop parallelism?

```
for(i=0; i<N; i++) {
   foo[i] = bar[i];
}
```

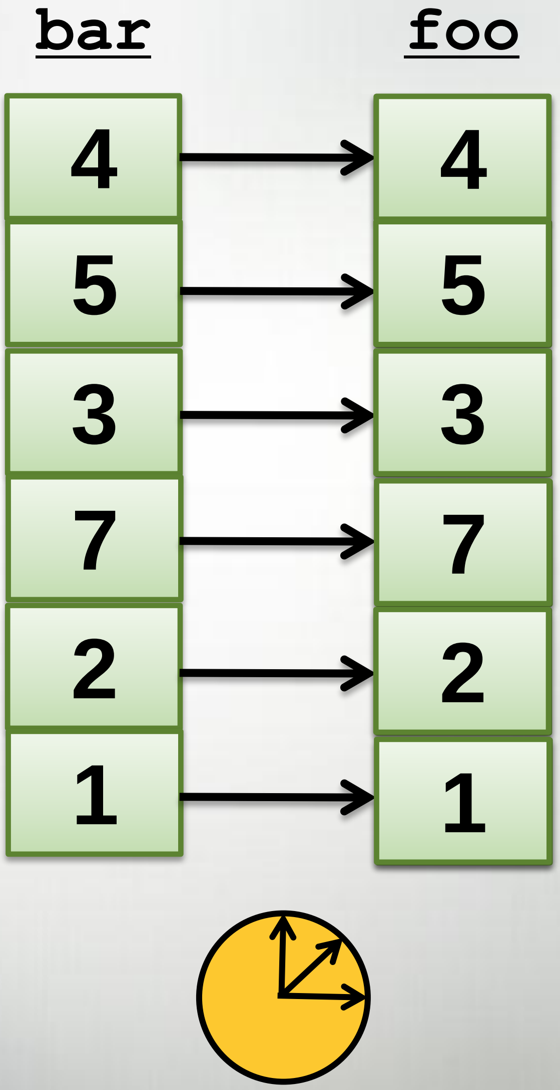
- A “normal” (non-parallelized) loop consists of a series of iterations that run one at a time (in order) on a single thread.



What is loop parallelism?

```
for(i=0; i<N; i++) {
   foo[i] = bar[i];
}
```

- A parallelized loop consists of a series of iterations that may run simultaneously on multiple threads.
- Every thread executes a distinct subset of the iterations



Outline

- Overview of the Cray XMT™ system
- Introduction to loop parallelism
  - Why do we care about loop parallelism?
  - Parallel execution of loops: threads and iterations
- **Identifying parallelism**
  - **Conditions necessary for parallelism**
  - **Compiler transformations to augment parallelism**
  - **Pragmas to assist parallelization**
- Implementing parallelism
  - Parallel regions
  - Single processor, multiprocessor, and loop future parallelism
- A parallelization example

When will the compiler parallelize a loop?

- The compiler attempts to parallelize your loops if:
  - It can figure out how to compute the number of iterations prior to executing the loop
  - It can prove that there are no dependences between iterations
  - There are no function calls with unknown side effects (e.g., output)
  - The loop has a simple structure (e.g., no multiple exits)
- Pragmas are promises made by the user that help the compiler establish that these conditions hold.

Example

- This loop parallelizes:

```
void foo(int n) {
     int i;
     int my_array[n];
     for (i = 0; i < n; i++) {
          my_array[i] = i;
     }
     return;
}
```

Example 2

- This loop does not:

```
void foo(int *a, int *b) {
     int i;
     for (i = 0; i < 10000; i++) {
          a[i] = b[i];
     }
}
```

- a and b may point to overlapping memory

```
foo(x+5000, x);
```

Using pragmas to help find parallelism

- The Cray XMT™ compiler supports a number of pragmas that can be used to give the compiler additional information about loops and the variables referenced inside them. The most commonly used are:
  - pragma mta assert noalias
  - pragma mta assert no dependence
  - pragma mta assert parallel

- The compiler treats these pragmas as promises by the user
  - The compiler trusts what you tell it
  - If you give incorrect information, and the compiler relies on it, your program may not run correctly.

The noalias pragma and restrict

```
void foo(int *x, int*y, int*z) {
  #pragma mta noalias *x, *y
  for (int i = 0; i < N; i++) {
    z[i] = x[i] + y[i];
  }
}
```

- Promises that the listed variables are not aliased with any other variables.
- Must appear within the scope and after the declarations of the listed variables.
- Only need to use once per variable (not once per loop).
- Can also use restrict pointers to get the same affect.

The no dependence pragma (or nodep)

```
#pragma mta assert noalias *IA
#pragma mta assert no dependence *IA
for (int i = 0; i < N; i++) {
  IA[i][1] = IA[i][INDEX[i]];
}
```

- Promises that any memory location accessed in the loop via any variable on the no dependence list is accessed by exactly one iteration of the loop
- Appears immediately before a loop
- Variables must be noalias or restrict pointers
- Can also use with no variable list. This makes the pragma apply to all memory references in the loop (and doesn’t require noalias pragmas).

The assert parallel pragma

```
#pragma mta assert parallel
for (int i = 0; i < N; i++) {
  printf("May appear out of order %d",i);
}
```

- Promises that the iterations of the loop can safely be executed concurrently without any synchronization.
- Does not force the compiler to parallelize the loop, but it is a strong suggestion.
- Should only be used when other techniques to get your loop to parallelize fail. It limits the types of optimizations and transformations the compiler can perform on the loop.
  - You are only asserting that the loop is parallel as written.
  - Compiler worries that loop transformations may invalidate that.

Compiler transformations for parallelism

- The compiler will attempt to restructure code to find or enhance parallelism:
    - Scalar expansion
    - Reductions
    - Loop collapse

- You can view the ways the compiler restructured your code in Canal (text-based) or in the Canal report of the Cray Apprentice2™ tool suite (GUI-based).

Scalar expansion

- This loop can not be parallelized as written because of dependences between the reads and writes of t in different iterations (writing t in one iteration may overwrite the value of t from another iteration before it is used):

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

Scalar expansion

- This loop can not be parallelized as written because of dependences between the reads and writes of t in different iterations (writing t in one iteration may overwrite the value of t from another iteration before it is used):

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

- The compiler solves this by converting the scalar integer t into an array of integers

Reductions

- The compiler attempts to recognize loops that calculate sums, products, minimums, and maximums over an array. E.g.:

```
int min = MAX_VAL;
for (i = 0; i < n; i++) {
  if (x[i] < min)
    min = x[i];
}
```

- The compiler converts these to reductions
  - Each thread computes the min/max/sum/product over a sub-section of the array.
  - Threads then combine results to determine the final value.

Nested parallelism

```
void foo(int* restrict num_bars, int size_x,
         int* restrict x, int* restrict bar)
{
  for (int i = 0; i < size_x; i++)
    for (int j = 0; j < num_bars[i]; j++)
      x[i] += bar[i + j];
}
```

- How do we handle nested parallel loops?
- Option 1: Go parallel for the outer loop, and then again for the inner loop.
  - Inefficient – there is a significant overhead to going parallel. If we nest, then every iteration of the outer loop has to pay that overhead.
  - Limits the effectiveness of the load balancing obtained by some of the scheduling methods.

Loop collapse

```
void foo(int* restrict num_bars, int size_x,
         int* restrict x, int* restrict bar)
{
  for (int i = 0; i < size_x; i++)
    for (int j = 0; j < num_bars[i]; j++)
      x[i] += bar[i + j];
}
```

- Option 2: Loop collapse.
  - Convert the nested pair of parallel loops to a single parallel loop that simulates the execution of the nested loops.
    - Create a new parallel loop to calculate the total number of iteration of the inner loop (across all iterations of the outer loop).
    - Convert the pair of loops into a single loop where each iteration corresponds to a distinct outer/inner iteration pair.
- Often a big performance win.

Collapse psuedocode

```
// t[i] = total # of inner loop iterations
// in first i iterations of outer loop
t[0] = 0;
for (i = 0; i < size_x; i++)
  t[i + 1] = t[i] + num_bars[i];

for (k = 0; k < t[size_x]; k++) {
  // Set i to index of largest element of t
  // less than k (use binary search)
  i = max_element_less_than(t,  k);
  j = k - t[i];

  x[i] += bar[i + j]; // original loop body
}
```

Outline

- Overview of the Cray XMT™ system
- Introduction to loop parallelism
  - Why do we care about loop parallelism?
  - Parallel execution of loops: threads and iterations
- Identifying parallelism
  - Conditions necessary for parallelism
  - Compiler transformations to augment parallelism
  - Pragmas to assist parallelization
- **Implementing parallelism**
  - **Parallel regions**
  - **Single processor, multiprocessor, and loop future parallelism**
- A parallelization example

Parallel regions

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

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

- The compiler will attempt to merge nearby loops, and intervening serial code, into parallel regions.
- A parallel region is surrounded by a single fork and join, saving the overhead of having to fork and join for every parallel loop.

The three forms of parallelism

- There are three forms of loop parallelism available: single processor, multiprocessor, and loop futures.
  - The compiler will choose one, based on estimates of overhead versus performance gained.
  - Compiler typically only chooses single processor or multiprocessor.
  - You can override the compiler’s choice with a pragma.
  - This is a per-region choice.

Single processor parallelism

```
#pragma mta loop single processor
for (int i = 0; i < small_size; i++)
  a[i] = b[i];
```

- Use multiple threads on a single processor.
- Very low overheard.
- Good for shorter loops where the time saved by going parallel does not justify the expense of more heavy-weight forms of parallelism.

Multiprocessor parallelism

```
#pragma mta loop multiprocessor
for (int i = 0; i < big_size; i++)
  a[i] = b[i];
```

- Use multiple threads on multiple processors.
- Higher overhead.
- Allows you to take advantage of all the resources of the machine.

Loop future parallelism

```
#pragma mta loop future
for (i = firstNode; i < lastNode; i++) {
  int nbr = Neighbors[i];
  int v = int_fetch_add(&Visited[nbr], 1);
  if (v == 0) BFS(nbr, A);
}
```

- Loop futures are a highly dynamic style of loop parallelism
    - For those familiar with futures, this is not just a loop of futures
    - Compiler still manages threads and schedules iterations
- Highest overhead form of loop parallelism
- The only form of parallelism where the number of assigned threads can increase dynamically
- Good for recursive-style loops with highly variable workloads

Outline

- Overview of the Cray XMT$^{\text{TM}}$ system
- Introduction to loop parallelism
  - Why do we care about loop parallelism?
  - Parallel execution of loops: threads and iterations
- Identifying parallelism
  - Conditions necessary for parallelism
  - Compiler transformations to augment parallelism
  - Pragmas to assist parallelization
- Implementing parallelism
  - Parallel regions
  - Single processor, multiprocessor, and loop future parallelism
- **A parallelization example**

An example

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

An example

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

An example

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

An example

```
     |   for (i = 0; i < n; i++) {
3 P:$|     if (b[i] == sought)
** 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];
```

An example

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

An example

```
     | #pragma mta assert noalias *a
     |   int i;
     |   int found_index = n;
     |   for (i = 0; i < n; i++) {
3 P:$|     if (b[i] == sought) {
** 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];
```

Summary

- Loop parallelism is an important technique for obtaining good performance on the Cray XMT^{TM} system.
- The compiler will automatically parallelize loop if it can establish that it is safe to do so.
    - Safe means that parallelization will preserve the correct program behavior.
- Pragmas may be used to assist the compiler in proving safety.
- The compiler will also attempt to aggressively transform loops to make them safe to parallelize.

CRAY

THE SUPERCOMPUTER COMPANY