CS4961 Parallel Programming

Lecture 6: More OpenMP, Introduction to Data Parallel Algorithms

Mary Hall
September 8, 2011

Homework 2: Due Before Class, Thursday, Sept. 8

'handin cs4961 hw2 <file>'

Problem 1: (Coherence) #2.15 in textbook

(a) Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable x in its cache, and it executes the assignment x=5. Finally, suppose that core 1 doesn't have x in its cache, and after core 0's update to x, core 1 tries to execute y=x. What value will be assigned to y? Why?

(b) Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to y? Why?

(c) Can you suggest how any problems you found in the first two parts might be solved?

Homework 2, cont.

Problem 2: (Bisection width/bandwidth)

(a) What is the bisection width and bisection bandwidth of a 3-d toroidal mesh.

(b) A planar mesh is just like a toroidal mesh, except that it doesn't have the wraparound links. What is the bisection width and bisection bandwidth of a square planar mesh.

Problem 3 (in general, not specific to any algorithm): How is algorithm selection impacted by the value of λ?

Homework 2, cont.

Problem 4: (λ concept) #2.10 in textbook

Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $10^{12}/p$ instructions, and each processor must send 10^9 (p-1) messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all its messages, and there won't be any delays due to things such as waiting for messages.

(a) Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?

(b) Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors?

Preview of Programming Assignment 1: Due Monday, Sept. 19

To be done on water.eng.utah.edu (you all have accounts - passwords available if your CS account doesn't work)

1. Write an average of a set of numbers in OpenMP for a problem size and data set to be provided. Use a block data distribution.
2. Write the same computation in Pthreads.

Report your results in a separate README file.

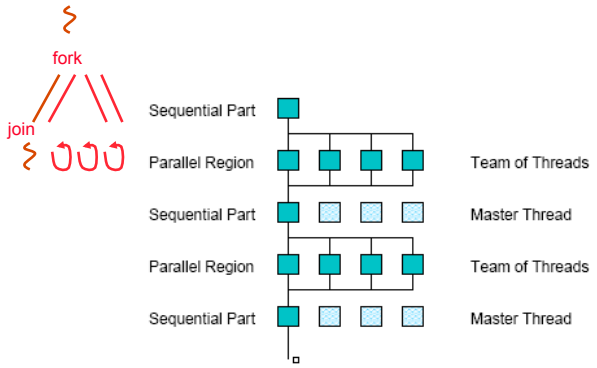
- What is the parallel speedup of your code? To compute parallel speedup, you will need to time the execution of both the sequential and parallel code, and report speedup = Time(seq) / Time (parallel)
- If your code does not speed up, you will need to adjust the parallelism granularity, the amount of work each processor does between synchronization points.
- Report results for two different numbers of threads.

Extra credit: Rewrite both codes using a cyclic distribution

Today's Lecture

- Data Parallelism in OpenMP
  - Expressing Parallel Loops
  - Parallel Regions (SPMD)
  - Scheduling Loops
  - Synchronization
- Sources of material:
  - Textbook
  - https://computing.llnl.gov/tutorials/openMP/

OpenMP Execution Model



OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the **same code redundantly (SPMD)**
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax

```
#pragma omp parallel [ clause [ clause ] ... ] new-line
  structured-block
```

- clause can include the following:

  `private` (list)

  `shared` (list)

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- PThreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private
- OpenMP:
  - **shared** variables are shared
  - **private** variables are private
  - Default is **shared**
  - Loop index is **private**

```
// shared, globals
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```

OpenMP Data Parallel Construct: Parallel Loop

- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning of Res
- Synchronization also automatic (barrier)

Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel Program:

```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

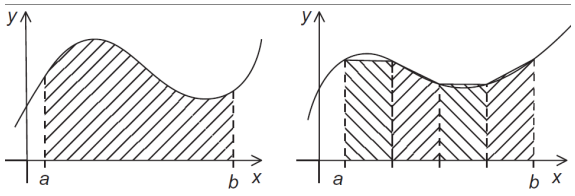

Limitations and Semantics

- Not all "element-wise" loops can be ||ized

```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {}
```

  - Loop index: signed integer
  - Termination Test: <,<=,>,=> with loop invariant int
  - Incr/Decr by loop invariant int; change each iteration
  - Count up for <,<=; count down for >,>=
  - Basic block body: no control in/out except at top
- Threads are created and iterations divvied up; requirements ensure iteration count is predictable
- What would happen if one thread were allowed to terminate early?

The trapezoidal rule



Serial algorithm

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```


OpenMp Reductions

- OpenMP has reduce operation

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i < 100; i++)   {
sum += array[i];
}
```

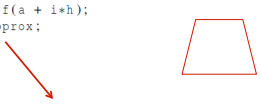
- Reduce ops and init() values (C and C++):

+	0	bitwise &	~0
-	0	bitwise |	0
*	1	bitwise ^	0

logical &	1
logical |	0

FORTRAN also supports min and max reductions


```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```



```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```


```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```


OpenMP critical directive

- Enclosed code
- executed by all threads, but
- **restricted to only one thread at a time**

`#pragma omp critical [ ( name ) ]` new-line

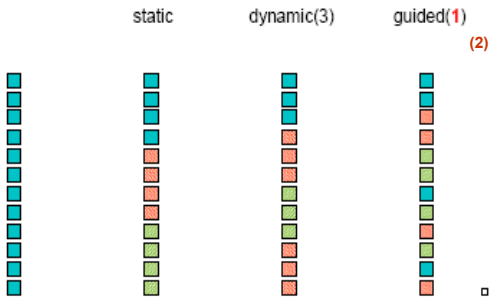
structured-block

- A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.
- All unnamed `critical` directives map to the same unspecified name.

Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - **`static([chunk])`** divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
  - **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default **`[chunk]`** is 1
  - **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

Loop scheduling



More loop scheduling attributes

- RUNTIME The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- AUTO The scheduling decision is delegated to the compiler and/or runtime system.
- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause (collapsed order corresponds to original sequential order).

Impact of Scheduling Decision

- Load balance
  - Same work in each iteration?
  - Processors working at same speed?
- Scheduling overhead
  - Static decisions are cheap because they require no run-time coordination
  - Dynamic decisions have overhead that is impacted by complexity and frequency of decisions
- Data locality
  - Particularly within cache lines for small chunk sizes
  - Also impacts data reuse on same processor

A Few Words About Data Distribution

- Data distribution describes how global data is partitioned across processors.
  - Recall the CTA model and the notion that a portion of the global address space is physically co-located with each processor
- This data partitioning is implicit in OpenMP and may not match loop iteration scheduling
- Compiler will try to do the right thing with static scheduling specifications

Common Data Distributions

- Consider a 1-Dimensional array to solve the global sum problem, 16 elements, 4 threads

CYCLIC (chunk = 1):

```
for (i = 0; i<blocksize; i++)
    … in [i*blocksize + tid];
```

BLOCK (chunk = 4):

```
for (i=tid*blocksize; i<(tid+1) *blocksize; i++)
    … in[i];
```

BLOCK-CYCLIC (chunk = 2):

The Schedule Clause

- Default schedule:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```


OpenMP Synchronization

- Implicit barrier
  - At beginning and end of parallel constructs
  - At end of all other control constructs
  - Implicit synchronization can be removed with `nowait` clause
- Explicit synchronization
  - `critical`
  - `atomic (single statement)`
  - `barrier`

Variation: OpenMP parallel and for directives

Syntax:

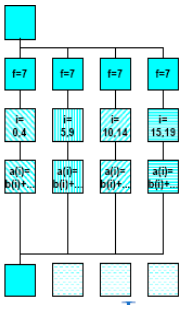
`#pragma omp for [ clause [ clause ] ... ] new-line`

for-loop

clause can be one of the following:

`shared (list)`
`private( list)`
`reduction( operator: list)`
`schedule( type [ , chunk ] )`
`nowait` (C/C++: on `#pragma omp for`)

```
#pragma omp parallel private(f) {
  f=7;
#pragma omp for
   for (i=0; i<20; i++)
   a[i] = b[i] + f * (i+1);
} /* omp end parallel */
```



OpenMP environment variables

`OMP_NUM_THREADS`

- sets the number of threads to use during execution
- when dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
- For example,

```
setenv OMP_NUM_THREADS 16  [csh, tcsh]
export OMP_NUM_THREADS=16  [sh, ksh, bash]
```

`OMP_SCHEDULE`

- applies only to `do/for` and `parallel do/for` directives that have the schedule type `RUNTIME`
- sets schedule type and chunk size for all such loops
- For example,

```
setenv OMP_SCHEDULE GUIDED,4  [csh, tcsh]
export OMP_SCHEDULE= GUIDED,4  [sh, ksh, bash]
```


Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```


OpenMP solution

```
      double sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) private(factor)
      for (k = 0; k < n; k++) {
         if (k % 2 == 0)
            factor = 1.0;
         else
            factor = -1.0;
         sum += factor/(2*k+1);
      }
```

Insures factor has private scope.

Summary of Lecture

- OpenMP, data-parallel constructs only
  - Task-parallel constructs later
- What's good?
  - Small changes are required to produce a parallel program from sequential (parallel formulation)
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor
- What is missing?
  - Not completely natural if want to write a parallel code from scratch
  - Not always possible to express certain common parallel constructs
  - Locality management
  - Control of performance