# CS4961 Parallel Programming

## Lecture 6:
## Data Parallelism in OpenMP, cont.
## Introduction to Data Parallel Algorithms

Mary Hall
September 9, 2010



## Homework 2, Due Friday, Sept. 10, 11:59 PM

- To submit your homework:
  - Submit a PDF file
  - Use the "handin" program on :
  - "handin cs4961 hw2 <prob2file>" the CADE machines
  - Use the following command

Problem 1 (based on #1 in text on p. 59):

Consider the Try2 algorithm for "count3s" from Figure 1.9 of p.19 of the text. Assume you have an input array of 1024 elements, 4 threads, and that the input data is evenly split among the four processors so that accesses to the input array are local and have unit cost. Assume there is an even distribution of appearances of 3 in the elements assigned to each thread which is a constant we call NTPT. What is a bound for the memory cost for a particular thread predicted by the CTA expressed in terms of $\lambda$ and NTPT.



## Homework 2, cont

Problem 2 (based on #2 in text on p. 59), cont.:

Now provide a bound for the memory cost for a particular thread predicted by CTA for the Try4 algorithm of Fig. 114 on p. 23 (or Try3 assuming each element is placed on a separate cache line).

Problem 3:

For these examples, how is algorithm selection impacted by the value of NTPT?

Problem 4 (in general, not specific to this problem):

How is algorithm selection impacted by the value of $\lambda$?



## Preview of Programming Assignment 1

- Write the prefix sum computation from HW1 in OpenMP for a problem size and data set to be provided. 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.
  - If possible, you should try different mappings to processors to find a version that achieves the best speedup.
  - What is the scheduling strategy you used to get the best speedup?

## Today's Lecture

- Data Parallelism in OpenMP
  - Expressing Parallel Loops
  - Parallel Regions (SPMD)
  - Scheduling Loops
  - Synchronization
- Sources of material:
  - Jim Demmel and Kathy Yelick, UCB
  - Allan Snavely, SDSC
  - Larry Snyder, Univ. of Washington
  - https://computing.llnl.gov/tutorials/openMP/



## Programming with Threads

Several Thread Libraries

- PTHREADS is the Posix Standard [IEEE std, 1995]
  - Solaris threads are very similar
  - Relatively low level
  - Portable but possibly slow
- OpenMP is newer standard [1997]
  - Support for scientific programming on shared memory architectures
- P4 (Parmacs) is another portable package [1987]
  - Higher level than Pthreads
  - http://www.netlib.org/p4/index.html



## A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races



## 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

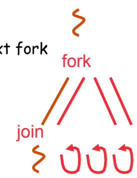
- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning of Res
- Synchronization also automatic (barrier)

| Serial Program: | Parallel Program: |
|---|---|
| <pre>void main()<br>{<br>   double Res[1000];<br><br>   for(int i=0;i<1000;i++) {<br>      do_huge_comp(Res[i]);<br>   }<br>}</pre> | <pre>void main()<br>{<br>   double Res[1000];<br>#pragma omp parallel for<br>   for(int i=0;i<1000;i++) {<br>      do_huge_comp(Res[i]);<br>   }<br>}</pre> |



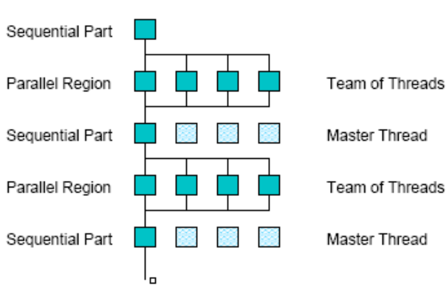
## OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  - Master thread creates team of threads
- Completion of a parallel construct:
  - Threads in the team synchronize -- **implicit barrier**
- Only master thread continues execution
- Implementation optimization:
  - Worker threads spin waiting on next fork






## OpenMP Execution Model






## OpenMP directive format C (also Fortran and C++ bindings)

- Pragmas, format

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

- Conditional compilation

```
#ifdef _OPENMP
   block,
   e.g., printf("%d avail.processors\n",omp_get_num_procs());
#endif
```

- Case sensitive
- Include file for library routines

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

## 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



## 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`



## 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 | logical & 1 |
| - 0 | bitwise \| 0 | logical \| 0 |
| * 1 | bitwise ^ 0 | |

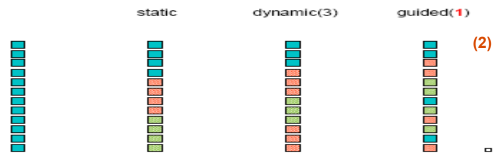
FORTRAN also supports min and max reductions



## 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. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.



## 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 (Ch. 5)

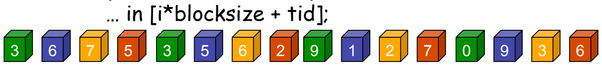
- Data distribution describes how global data is partitioned across processors.
  - Recall the CTA model and the notion that a portion of the global address space is physically co-located with each processor
- This data partitioning is implicit in OpenMP and may not match loop iteration scheduling
-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

## Common Data Distributions

- Consider a 1-Dimensional array to solve the count3s problem, 16 elements, 4 threads

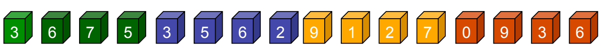
CYCLIC (chunk = 1):

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

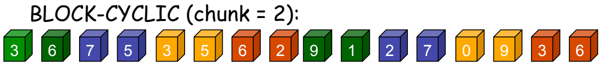



BLOCK (chunk = 4):

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




BLOCK-CYCLIC (chunk = 2):






## 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.



## Variation: OpenMP parallel and for directives

**Syntax:**

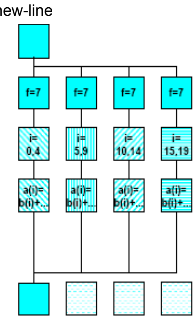
`#pragma omp for` [ clause [ clause ] ... ] new-line

*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 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)

## 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]
```



## OpenMP runtime library, Query Functions

`omp_get_num_threads`:

Returns the number of threads currently in the team executing the parallel region from which it is called

```
int omp_get_num_threads(void);
```

`omp_get_thread_num`:

Returns the thread number, within the team, that lies between `0` and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread `0`

```
int omp_get_thread_num(void);
```



## 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