CMSC 714
Lecture 5
OpenMP

Alan Sussman

Notes

- MPI project posted
  - due 2 weeks from Thursday, Sept. 28
  - Send any questions about project spec, or running on zaratan cluster, to me or Keon, or post on Piazza
- Don't forget to send questions for readings
  - additional readings posted last week, with who should send questions (check again, since class roster has changed)
- We will finish OpenMP next class
  - Other readings/lectures pushed back – see Readings web page

OpenMP

- Support Parallelism for SMPs
  - provide a simple portable model
  - allows both shared and private data
  - provides parallel do loops
- Includes
  - automatic support for fork/join parallelism
  - reduction variables
  - atomic statement
    - one process executes at a time
  - plus a lot more

OpenMP

- Characteristics
  - Both thread-local & shared memory (depending on directives)
  - Parallelism : directives for parallel loops, functions
  - Compilers convert programs into multi-threaded (i.e. pthreads)
  - Not able to run on more than one node in a cluster
- Example

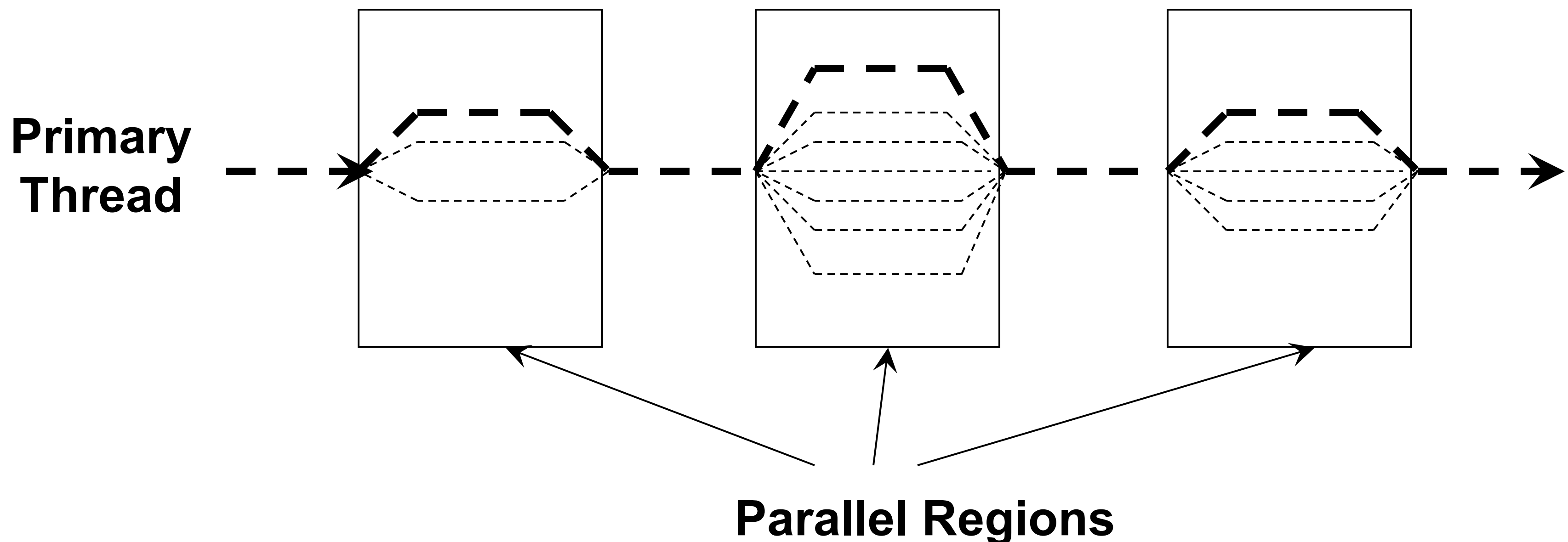
```
#pragma omp parallel for private(i)
for (i=0; i<NUPDATE; i++) {
	int ran = random();
	table[ ran & (TABSIZE-1) ] ^= stable[ ran >> (64-LSTSIZE) ];
}
```

More on OpenMP

- Characteristics
  - Not a full parallel language, but a language extension
  - A set of standard compiler directives and library routines
  - Used to create parallel Fortran, C and C++ programs
  - Usually used to parallelize loops
  - Standardizes last ~20 years of SMP practice
- Implementation
  - C/C++ compiler directives using #pragma omp <directive>
  - Parallelism can be specified for regions & loops
  - Data can be
    - Private – each thread has local copy
    - Shared – single copy for all threads

OpenMP – Programming Model

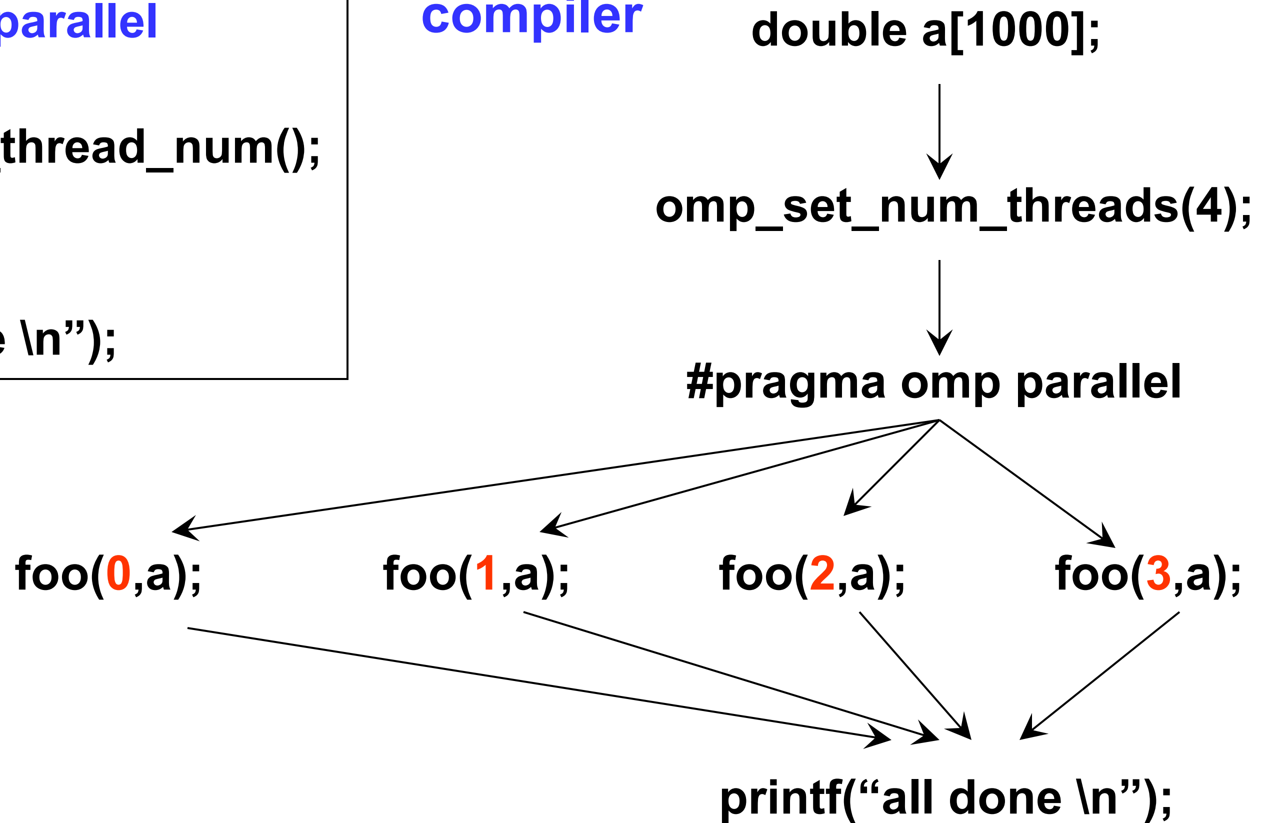
- Fork-join parallelism (restricted form of MIMD)
  - Normally single thread of control (primary)
  - Worker threads spawned when parallel region encountered
  - Barrier synchronization required at end of parallel region



OpenMP – Example Parallel Region

- Task level parallelism – #pragma omp parallel { … }

```
double a[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int id = omp_thread_num();
  foo(id,a);
}
printf("all done \n");
```



OpenMP – Example Parallel Loop

- **Loop level parallelism – #pragma omp parallel for**
  - **Loop iterations are assigned to threads, invoked as functions**

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

```
#pragma omp parallel
{
    int id, i, nthreads,start, end;
    id = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    start = id * N / nthreads ;        // assigning
    end = (id+1) * N / nthreads ;      // work
    for (i=start; i<end; i++) {
        foo(i);
    }
}
```

Race conditions when threads interact

- Unintended sharing of variables can lead to race conditions
- Race condition: program outcome depends on the scheduling order of threads
- How can we prevent data races?
  - Use synchronization
  - Change how data is stored

OpenMP details

OpenMP pragmas

- Pragma: a compiler directive in C or C++
- Mechanism to communicate with the compiler
- Compiler may ignore pragmas

```
#pragma omp construct [clause [clause] ... ]
```

Hello World in OpenMP

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

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

- Compiling:

```
gcc -fopenmp hello.c -o hello
```

- Setting number of threads:

```
export OMP_NUM_THREADS=2
```

Parallel for

- Directs the compiler that the immediately following for loop should be executed in parallel

```
#pragma omp parallel for [clause [clause] ... ]
for (i = init; test_expression; increment_expression) {
    ...
    do work
    ...
}
```

Parallel for example

- saxpy (single precision a*x+y) example

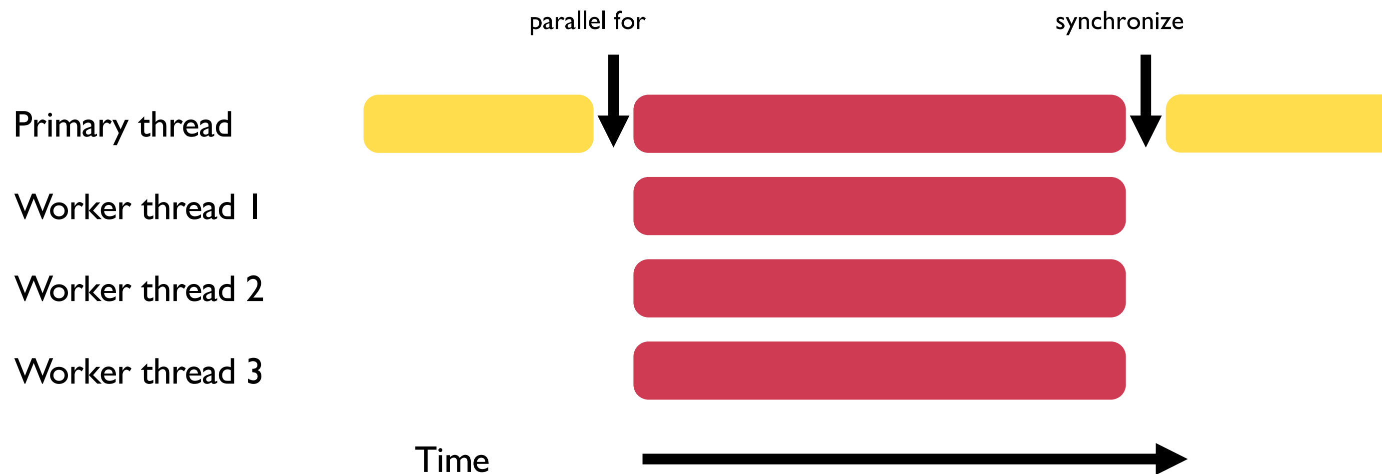
```
int main(int argc, char **argv)
{
    ...

#pragma omp parallel for
    for (int i = 0; i < n; i++) {
        z[i] = a * x[i] + y[i];
    }

    ...
}
```

Parallel for execution

- Primary thread creates worker threads
- All threads divide iterations of the loop among themselves



Number of threads

- Use environment variable in shell

```
export OMP_NUM_THREADS=X
```

- Use omp_set_num_threads(int num_threads)
  - Set the number of OpenMP threads to be used in parallel regions
- int omp_get_num_procs(void)
  - Returns the number of available processors/cores
  - Can be used to decide the number of threads to create

Data sharing defaults

- Most variables are shared by default
- Global variables are shared
- Exception: loop index variables are private by default
- Stack variables in function calls from parallel regions are also private to each thread (thread-private)

Overriding defaults using clauses

- Specify how data is shared between threads executing a parallel region
- private(list)
- shared(list)
- default(shared | none)
- reduction(operator: list)
- firstprivate(list)
- lastprivate(list)

https://www.openmp.org/spec-html/5.0/openmpsu106.html#x139-5540002.19.4

firstprivate clause

- Initializes each thread's private copy to the value of the primary thread's copy

```
val = 5;

#pragma omp parallel for firstprivate(val)
for (int i = 0; i < n; i++) {
    ... = val + 1;
}
```

lastprivate clause

- Writes the value belonging to the thread that executed the last iteration of the loop to the primary's copy
- Last iteration determined by sequential order

```
#pragma omp parallel for lastprivate(val)
for (int i = 0; i < n; i++) {
    val = i + 1;
}

printf("%d\n", val);
```

reduction(operator: list) clause

- Reduce values across private copies of a variable
- Operators: +, -, *, &, |, ^, &&, ||, max, min
  - User-defined operators can be created

```
#pragma omp parallel for reduction(+: val)
for (int i = 0; i < n; i++) {
    val += i;
}

printf("%d\n", val);
```

https://www.openmp.org/spec-html/5.0/openmpsu107.html#x140-5800002.19.5

Loop scheduling

- Assignment of loop iterations to different worker threads
- Default schedule tries to balance iterations among threads
- User-specified schedules are also available

User-specified loop scheduling

- Schedule clause

```
schedule (type[, chunk])
```

- type: static, dynamic, guided, runtime
- static: iterations divided as evenly as possible (#iterations/#threads)
  - chunk size < #iterations/#threads can be used to interleave threads
- dynamic: assign a chunk size block to each thread
  - When a thread is finished, it retrieves the next block from an internal work queue, so requires a scheduler thread
  - Default chunk size = 1

Other schedules

- guided: similar to dynamic but start with a large chunk size and gradually decrease it for handling load imbalance between iterations
- auto: scheduling delegated to the compiler
- runtime: use the OMP_SCHEDULE environment variable

https://software.intel.com/content/www/us/en/develop/articles/openmp-loop-scheduling.html

Calculate the value of $\pi = \int_0^1 \frac{4}{1+x^2}$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;

    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

Calculate the value of $\pi = \int_0^1 \frac{4}{1+x^2}$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;
    h = 1.0 / (double) n;
    sum = 0.0;

    #pragma omp parallel for private(x) reduction(+: sum)
    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

Synchronization

- Concurrent access to shared data may result in inconsistencies
- Use mutual exclusion to avoid that
- critical directive
- atomic directive
- Library lock routines

https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/appendix/adding-parallelism-to-your-program/replacing-annotations-with-openmp-code/adding-openmp-code-to-synchronize-the-shared-resources.html