Advanced Topics in Numerical Analysis: High Performance Computing
MATH-GA 2012.003 and CSCI-GA 2945.003

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

Spring 2015, Monday, 5:10–7:00PM, WWH #202

April 6, 2015
Outline

Organization issues

Shared memory parallelism–OpenMP
Organization and today’s class

Organization:

- Homework #1 solution posted. Stampede? Anybody?
- Homework #3 will be posted tonight (sorry for the delay).
Organization and today’s class

Organization:
- Homework #1 solution posted. Stampede? Anybody?
- Homework #3 will be posted tonight (sorry for the delay).

Today:
- Distributed memory programming (OpenMP).
- Visualization with Paraview
Plan for the rest of this course

Your input is welcome

- **Programming**: OpenCL (for GPGPUs and MICs); Performance with accelerators;
- **Tools**: intro to shell scripting; Paraview visualization; (Partitioning with Metis)
- **Algorithms**: Multigrid and FMM
Outline

Organization issues

Shared memory parallelism–OpenMP
Shared memory programming model

- Program is a collection of control threads, that are created dynamically
- Each thread has private and shared variables
- Threads can exchange data by reading/writing shared variables
- **Danger:** more than 1 processor core reads/writes to a memory location: race condition

Main difference to MPI is that only one process is running, which can fork into shared memory threads.
A **process** is an independent execution unit, which contains their own state information (pointers to instruction and stack). One process can contain several threads.

**Threads** within a process share the same address space, and communicate directly using shared variables. Separate stack but shared heap memory.

Using several threads can also be useful on a single processor (**“multithreading”**), depending on the memory latency.
Shared memory programming—Literature

OpenMP standard online:
www.openmp.org

Very useful online course:
www.youtube.com/user/OpenMPARB/

Recommended reading:

Chapter 6 in

Chapter 7 in
Shared memory programming with OpenMP

- Split program into serial and into parallel regions
- Race condition: more than one thread reads/writes to the same shared memory location
- Easy to parallelize loops without data dependencies by adding #pragma commands
- Pragmas are compiler directive externals to the programming language; they are ignored by the compiler if it doesn’t understand them
Shared memory programming with OpenMP

- Provides synchronization functions such as the collective calls in MPI
- Tell the compiler which parts are parallel
- Compiler generates threaded code; usually it defaults to one (or 2/4 for hyper-threading) threads per core
- You can set the number of threads used through the environment variable OMP_NUM_THREADS or inside the code using the library call `omp_set_num_threads(4)`
- Dependencies in parallel parts requires synchronization between threads
Shared memory programming with OpenMP

```c
#pragma omp parallel num_threads(3)
{
    // ...
}
```

Define number of threads explicitly

Can be done through environment variable:

- `export OMP_NUM_THREADS=8` (bash)
- `setenv OMP_NUM_THREADS 8` (csh)
#pragma omp parallel for default(none)
    private() shared()
for (i=0; i<N; ++i) {
    // ....
}

Per default, all variables are shared in parallel part
Can explicitly specify which variables are private and which shared
private(): Non-initialized variable
firstprivate(): initialized with outside value
lastprivate(): main thread returns with value from parallel construct
Shared memory programming with OpenMP

```c
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=0; i<N; ++i) {
        // so something ...
    }
    #pragma omp for
    for (i=0; i<N; ++i) {
        // do something else ...
    }
}
```

Threads will not synchronize at the end of parallel part
Shared memory programming with OpenMP

Race condition

for (i=0; i < N; i++)
    a[i] = b[i] + c[i];

for (i=0; i < N; i++)
    d[i] = a[i] + b[i];
result = 0;
#pragma omp parallel for reduction(+:result)
for (i = 1; i < N, ++i) {
    result += a[i];
}

Reduction is analogue to MPI_Reduce
Other reduction operations: −,∗,max,min,...
int main (int argc, char **argv) {
    /* initialize variables, allocate and fill vectors */

    printf("Hello hello. I’m thread %d of %d\n", 
        omp_get_thread_num(), omp_get_num_threads());
}
for (p = 0; p < passes; ++p)
{

    for (i = 0; i < n; ++i) {
        c[i] = a[i] * b[i];
    }
    prod = 0.0;

    for (i = 0; i < n; ++i) {
        prod += c[i];
    }
}
/* free memory and return */
int main ( int argc, char **argv) {

    /* initialize variables, allocate and fill vectors */
    #pragma omp parallel
    { printf("Hello hello. I’m thread %d of %d\n", 
            omp_get_thread_num(), omp_get_num_threads());
    }
    for (p = 0; p < passes; ++p)
    {
        #pragma omp parallel for default(none) shared(n,a,b,c)
            for (i = 0; i < n; ++i) {
                c[i] = a[i] * b[i];
            }
        prod = 0.0;
        #pragma omp parallel for reduction(+:prod)
            for (i = 0; i < n; ++i) {
                prod += c[i];
            }
    }

    /* free memory and return */
}
Shared memory programming with OpenMP

```c
#pragma omp sections [...] private() 
    reduction() nowait...
{
    #pragma omp section ...
        structured block
#pragma omp section ...
        another structured block
}
```

implied barrier at the end of SECTIONS directive

sections are split amongst threads; some threads get none or more than one section

no branching in and out of sections
Shared memory programming with OpenMP

Example

```c
#pragma omp parallel shared(n,a,b,x,y),
    private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0;i<n;i++)
            b[i] += a[i];

        #pragma omp section
        for (i=0;i<n;i++)
            x[i] = sin(y[i]);
    }
}
```
Shared memory programming with OpenMP

```c
#define omp single [...] private()
   reduction() nowait...
{
   structured block
}
```

only one thread in the team executes the enclosed block

useful for code in parallel environment that isn’t thread-safe (such as input/output code)

rest of threads wait at the end of enclosed block (unless NOWAIT clause is used)
Shared memory programming with OpenMP

More clauses

- Implicit **barrier** at the end of parallel region. Can be called explicitly using `#pragma omp barrier`.
- **critical** sections: only one thread at a time in critical region with the same name: `#pragma omp critical [name]`.
- **atomic** operation: protects updates to individual memory location. Only single expression allowed `#pragma omp atomic`.
- **locks**: low-level library routines
Shared memory programming with OpenMP

Critical Example

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
        task = produce_task();
        #pragma omp critical (task_queue)
        {
            insert_into_queue(task);
        }
    }
    #pragma omp section
    {
        #pragma omp critical (task_queue)
        {
            task = delete_from_queue(task);
        }
    }
}
```
#pragma omp parallel shared(n, ic) private(i) {
    for (i = 0; i < n; ++i) {
        // do some stuff
        #pragma omp atomic
        ic = ic + 1;
    }
}

#pragma omp parallel shared(n, ic) private(i) {
    for (i = 0; i < n; ++i) {
        // do some stuff
        #pragma omp atomic
        ic = ic + F();
    }
}

Computation of F is not atomic, only increment.
allowable atomic operations:
x += expr   x++   x--
Locks control access to shared variables.

- Lock variables must be accessed only through locking routines:
  ```
  omp_init_lock   omp_destroy_lock
  omp_set_lock    omp_unset_lock
  omp_test_lock
  ```

- A lock is a type `omp_lock_t`

- `omp_set_lock (omp_lock_t *lock)`
  forces calling thread to wait until the specified lock is available
```
omp_lock_t my_lock;

int main() {
    omp_init_lock(&my_lock);
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int i, j;
        for (i = 0; i < 10; ++i) {
            omp_set_lock(&my_lock);
            printf("Thread %d -- starting lock region\n", tid);
            ...
            printf("Thread %d -- finishing lock region\n", tid);
            omp_unset_lock(&my_lock);
        }
    }
    omp_destroy_lock(&my_lock);
}
```
Shared memory programming with OpenMP

Demos:
https://github.com/NYU-HPC15/lecture8
Visualization with Paraview

Download Paraview:
http://www.paraview.org/

Paraview Tutorial:
http://www.paraview.org/Wiki/The_ParaView_Tutorial

Example files are checked in /vtkdata in
https://github.com/NYU-HPC15/lecture8