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

Mar. 30, 2015
Outline

Organization issues

Last week summary

Debugging and profiling

Shared memory parallelism–OpenMP
Organization and today’s class

Organization:

- Final projects will be due in final’s week or the week after.
- Homework #3 will be posted during this week.
- I’m (almost) done with checking homeworks #0 and #1.
Organization and today’s class

Organization:

- Final projects will be due in final’s week or the week after.
- Homework #3 will be posted during this week.
- I’m (almost) done with checking homeworks #0 and #1.

Today:

- MPI bug examples
- Distributed memory programming (OpenMP).
Comments for homeworks #0 and #1

What's to say about... 

- `pow(x, 2)` for $x^2$, `pow(x, 0.5)` for $\sqrt{x}$
- Do not allocate storage for the full sparse matrix $A$, which only contains two nonzero entries!
- No need to print vectors or other long lists of doubles!
- In loops: quantities that do not change, should be computed once outside the loop (it’s possible that the compiler optimization tries to fix that).
Comments/tips for homeworks

Some more good practice comments...

- Remove/comment out unused variables (avoids warnings)
- Files to be checked into repos such as git: Mainly text-based source files such as *.tex, *.c, *.cpp, if necessary *.pdf, *.png. Avoid to add executables, log-files (*.log, *.aux), compiled \LaTeX\ files (*.pdf, *.dvi) etc.
- Use as little communication as possible. The most interesting questions in parallel HPC are:
  - What’s the best method/algorithm?
  - What’s the best way to parallelize it? Intellectual challenge is to understand the algorithm and to find the concurrency/parallelism in the algorithm.
  - How can I use as little communication as possible?
  - What is the cleanest and most efficient way to implement this?
Outline

Organization issues

Last week summary

Debugging and profiling

Shared memory parallelism—OpenMP
Submitting jobs on Stampede

Overview of HPC cluster
Submitting jobs on Stampede

Stampede user guide: https://portal.tacc.utexas.edu/user-guides/stampede

Batch facilities: SGE, LSF, SLURM. Stampede uses SLURM, and these are some of the basic commands:

- submit/start a job: sbatch jobscript
- see status of my job: squeue -u USERNAME
- cancel my job: scancel JOBID
- see all jobs on machine: showq | less
Submitting jobs on Stampede
Example job script (in git repo for lecture5)

#!/bin/bash
#SBATCH -J myMPI  # job name
#SBATCH -o myMPI.o  # output and error file name
#SBATCH -n 32  # total number of mpi tasks
#SBATCH -p development  # queue -- normal, development, etc.
#SBATCH -t 01:30:00  # run time (hh:mm:ss) - 1.5 hours
#SBATCH --mail-user=username@tacc.utexas.edu
#SBATCH --mail-type=begin  # email me when the job starts
#SBATCH --mail-type=end  # email me when the job finishes
ibrun ./a.out  # run the MPI executable
Outline

Organization issues

Last week summary

Debugging and profiling

Shared memory parallelism—OpenMP
Debugging/profiling tools

- printf — fishing for bug, but sometimes useful
- gdb — GNU debugger: serial, but can also be attached to a parallel task
- Totalview (commercial debugger/profiler, available on many HPC resources)
- DDT (another commercial debugger/profiler; available on Stampede)
- TAU: Tuning and Analysis Utility
- PAPI: Performance Application Programming Interface
- HPCtoolkit/perfexpert: Suite of (open source) analysis and profiling tools
- valgrind (say: “val-grinned”) and cachegrind: memory/cache debugger and profiler
Valgrind and cachegrind

Valgrind

- memory management tool and suite for debugging, also in parallel
- profiles heap (not stack) memory access
- simulates a CPU in software
- running code with valgrind makes it slower by factor of 10-100
- only available in development version on latest version of Mac OS

Documentation: http://valgrind.org/docs/manual/

- memcheck
  - finds leaks
  - inval. mem. access
  - uninitialize mem.
  - incorrect mem. frees

- cachegrind
  - cache profiler
  - sources of cache misses

- callgrind
  - extension to cachegrind
  - function call graph
Outline

- Organization issues
- Last week summary
- Debugging and profiling
- 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.
Threads versus process

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

- **POSIX Threads** (Pthreads) library; more intrusive than OpenMP.
- **PGAS languages**: partitioned global address space: logically partitioned but can be programmed like a global memory address space (communication is taken care of in the background)
- **OpenMP**: open multi-processing is a light-weight application interface (API), that supports shared memory parallelism.
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

#include <omp.h>

int main () {
    // initialization
    // serial code; thread 0
    #pragma omp parallel private(a,b) shared(c,d)
    {
        // parallel section; threads numbered 0,1,...
    }
    // resume serial code; only thread 0

    ▶ implicit barrier at the end of the parallel part
    ▶ if a thread terminates in the parallel part, all other threads will terminate as well
    ▶ parallel constructs within a parallel region are possible
Shared memory programming with OpenMP

- Provides synchronisation 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 hyperthreading) 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 directive [clause list]
```

Created teams of threads
Specify work sharing
Declare private/shared variables
Exclusive execution (critical regions)
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)
Shared memory programming with OpenMP

```c
#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

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

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

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

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

**Barrier Region**

```
<table>
<thead>
<tr>
<th>idle</th>
<th>idle</th>
<th>idle</th>
</tr>
</thead>
</table>
```

```
| idle | idle | idle |
```

```
| idle | idle | idle |
```

**Wait!**

**Barrier**

**Time**
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

Demos:
https://github.com/NYU-HPC15/lecture7