Introduction to Parallel and Distributed Processing

Introduction to OpenMP

Jaroslaw ‘Jaric’ Zola

http://www.jzola.org/

Suggested Reading

- OpenMP reference guide http://openmp.org/wp/
- OpenMP tutorial
  https://computing.llnl.gov/tutorials/openMP/

OpenMP Standard

- Programming standard for shared memory maintained by the industry consortium, support for C/C++ and Fortran
- Built around compiler directives, supporting run-time library and environment variables
- Focus on simplification of thread-based parallelism
- Based on fork-join paradigm, new standard directly addresses data parallelism

OpenMP Support

- All modern compilers provide strong support for OpenMP
- Using in GCC:

```
g++ -std=c++11 -fopenmp -O3 ...
```

OpenMP General Idea

- Instrument code with “pragmas”, compiler behavior modifiers
- Mark regions that should be executed by a team of threads
- Mark level of parallelism, shared variables and synchronization

```
#include <iostream>

int main(int argc, char* argv[]) {
#pragma omp parallel
  {
    std::cout << "hello world" << std::endl;
  }
  return 0;
} // main
```

OpenMP Syntax Basics

- Most of the OpenMP directives are of the form:
  `#pragma omp` *construct [clause [[,] clause]...] new-line*
  structured-block | for-loop

- `#pragma omp parallel` construct forms a team of threads

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

int main(int argc, char* argv[]) {
#pragma omp parallel num_threads(2)
  {
#pragma omp parallel num_threads(2)
    {
      if (omp_get_thread_num() == 0) {
        std::cout << omp_get_num_threads();
      }
    }
  }
  return 0;
} // main
```

Basic Environment Variables

- `OMP_NUM_THREADS`
  Default number of threads to use in parallel regions
- `OMP_THREAD_LIMIT`
  Total number of threads allowed
- `OMP_NESTED`
  Enable/disable nested parallel regions

Work Sharing Constructs

- Work sharing constructs:
  `for`, `sections`, `single`, `master`, `task`
- Must be encountered by all threads in a team
- Work is distributed over all threads
- A work-sharing construct does not spawn any additional threads

Data Sharing Constructs

- `shared` – variable is shared between threads
- `private` – variable is private to each thread, i.e. each thread has a local copy, which is not initialized, and not passed outside of parallel region
- `firstprivate` – like `private`, but initialized
- `lastprivate` – like `private`, but the original copy updated after the construct

Data Sharing Rules

- The `for` loop iteration variable is `private`
- Automatic variables inside the parallel construct are `private`
- Variables with heap allocated storage are `shared`
- Static data members are `shared`
- Static variables declared in the parallel construct are `shared`
- Consts of type without mutable member are `shared`

OpenMP Synchronization

- Threads are synchronized implicitly at the beginning and at the end of parallel region and work sharing constructs (exception `master` construct and `nowait` clause)
- Threads can be explicitly synchronized (`barrier`, `critical`, `atomic`)
- Synchronization implies memory flushes

OpenMP Synchronization

- `critical` at any point of time only one thread can be executing region (critical section)
- `atomic` works only with expression statements, allows compiler to exploit atomic operations supported by CPU
- `barrier` specifies a point in the execution where all threads in a team wait for each other

Code Examples

- x shared by all threads executing loop

```
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
  int N = 1024 * 1024 * 1024;
  std::vector<int> x(N);

#pragma omp parallel for shared(x)
  for (int i = 0; i < N; ++i) x[i] = i;

  return 0;
} // main
```

Code Examples

- What will happen if x is not `firstprivate`

```
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
  int N = 1024 * 1024;
  std::vector<int> x(N, -1);

#pragma omp parallel
  {
    #pragma omp for firstprivate(x) nowait
      for (int i = 0; i < N; ++i) x[i] = i;

    #pragma omp critical
      {
        std::cout << x[0] << std::endl;
      }
  }
  return 0;
} // main
```

Code Examples

```
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
  int N = 1024 * 1024;
  std::vector<int> x(N, -1);

#pragma omp parallel shared(x)
  {
    #pragma omp for nowait
      for (int i = 0; i < N; ++i) x[i] = i;

    #pragma omp barrier
      std::cout << "wow" << std::endl;
  }
  return 0;
} // main
```

Parallel For

- `schedule` – how loop should be distributed (static, dynamic, guided, auto, runtime)
- `reduction(operator:var)` – use `var` to perform reduction with `operator`

```
#include <iostream>
#include <vector>

int main(int argc, char* argv[]) {
  double S = 0.0;
  int N = 1024 * 1024 * 1024
  std::vector<double> x(N, 0.1);

#pragma omp parallel for shared(x) schedule(auto) \
  reduction(+:S)
  for (int i = 0; i < N; ++i) S += x[i];

  return 0;
} // main
```

Parallel Single & Master

- `single` – one thread executes the block, other wait
- `master` – only master executes the block, other proceed
- `nowait` – removes barrier from given block

```
#include <iostream>
#include <omp.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
#pragma omp parallel
  {
#pragma omp single nowait
    {
      sleep(1);
      std::cout << "single: " << omp_get_thread_num() << std::endl;
    }
#pragma omp master
    {
      std::cout << "master: " << omp_get_thread_num() << std::endl;
    }
  }
  return 0;
} // main
```

Parallel Sections

- Each section executed by one thread
- If more sections than threads some threads execute multiple sections
- If more threads than sections idle threads wait (unless `nowait`)

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

int main(int argc, char* argv[]) {
#pragma omp parallel sections
  {
#pragma omp section
    std::cout << "A: " << omp_get_thread_num() << std::endl;

#pragma omp section
    std::cout << "B: " << omp_get_thread_num() << std::endl;
  }
  return 0;
} // main
```

Fibonacci Again

```
#include <iostream>

int fib(int n) {
  int i, j;
  if (n < 2) return n;
#pragma omp parallel sections shared(i, j) num_threads(2)
    {
#pragma omp section
      i = fib(n - 1);
#pragma omp section
      j = fib(n - 2);
    }
  return i + j;
} // fib

int main(int argc, char* argv[]) {
  int x = fib(64);
  return 0;
} // main
```

Parallel Tasks

- `task` construct defines a task to be executed by a thread
- `taskwait` waits for completion of all **child** tasks

```
#include <iostream>

int fib(int n) {
  int i, j;
  if (n < 2) return n;
#pragma omp task shared(i)
    i = fib(n - 1);

#pragma omp task shared(j)
    j = fib(n - 2);

#pragma omp taskwait
  return i + j;
} // fib

int main(int argc, char* argv[]) {
#pragma omp parallel
#pragma omp single nowait
  std::cout << fib(36);
  return 0;
} // main
```

Parallel Tasks

- `depend` provides mechanism for defining task dependencies
- `untied` makes task independent of threads

```
int fib(int n) {
  int i, j, S;
  if (n < 2) return n;

#pragma omp task shared(i) depend(out:i)
    i = fib(n – 1);

#pragma omp task shared(j) depend(out:j)
    j = fib(n – 2);

#pragma omp task shared(i, j, S) depend(in:i,j) untied
    S = i + j;

#pragma omp taskwait
  return S;
} // fib

int main(int argc, char* argv[]) {
#pragma omp parallel
#pragme omp single nowait
  fib(36);
  return 0;
} // main
```

For Fun

- Implement Cilk+ examples using OpenMP tasks