Parallelization with MPI and OpenMP and discussions of project 1

Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no[1,2]

Department of Physics and Center fo Computing in Science Education, University of Oslo, Oslo, Norway[1]

Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA[2]

March 22, 2024

© 1999-2024, Morten Hjorth-Jensen Email morten.hjorth-jensen@fys.uio.no. Released under CC Attribution-NonCommercial 4.0 license

Plans for the week of March 18-22

Topics

- Discussion of project 1 and possible alternatives for project 2
- Wrap up of parallelization discussions, see also slides from last week
- Note change of deadline for project 1 to April 1.

Teaching Material, videos and written material

- Background literature: Using OpenMP by Chapman et al. and Using MPI by Gropp et al..

Alternatives for project 2

1. Fermion VMC, continuation of project 1
2. Deep learning applied to project 1, neural networks/Boltzmann machines
3. Hartree-Fock theory and time-dependent theories
4. Many-body methods like coupled-cluster theory or other many-body methods
5. Quantum computing and possibly quantum machine learning
6. Suggestions from you

What is OpenMP

- OpenMP provides high-level thread programming
- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a fork-join pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
- The newly spawned threads work simultaneously with the master thread
- At the end of a parallel region, the new threads are destroyed

Many good tutorials online and excellent textbook

1. Using OpenMP, by B. Chapman, G. Jost, and A. van der Pas
2. Many tutorials online like OpenMP official site

Getting started, things to remember

- Remember the header file

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

- Insert compiler directives in C++ syntax as

```
#pragma omp...
```

- Compile with for example *c++ -fopenmp code.cpp*
- Execute
  - Remember to assign the environment variable **OMP NUM THREADS**
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten

OpenMP syntax

- Mostly directives

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

- Some functions and types

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

- Most apply to a block of code
- Specifically, a **structured block**
- Enter at top, exit at bottom only, exit(), abort() permitted

Different OpenMP styles of parallelism

OpenMP supports several different ways to specify thread parallelism

- General parallel regions: All threads execute the code, roughly as if you made a routine of that region and created a thread to run that code
- Parallel loops: Special case for loops, simplifies data parallel code
- Task parallelism, new in OpenMP 3
- Several ways to manage thread coordination, including Master regions and Locks
- Memory model for shared data

General code structure

```c
#include <omp.h>
main ()
{
int var1, var2, var3;
/* serial code */
/* ... */
/* start of a parallel region */
#pragma omp parallel private(var1, var2) shared(var3)
{
/* ... */
}
/* more serial code */
/* ... */
/* another parallel region */
#pragma omp parallel
{
/* ... */
}
}
```

Parallel region

- A parallel region is a block of code that is executed by a team of threads
- The following compiler directive creates a parallel region

```
#pragma omp parallel { ... }
```

- Clauses can be added at the end of the directive
- Most often used clauses:
  - **default(shared)** or **default(none)**
  - **public(list of variables)**
  - **private(list of variables)**

Hello world, not again, please!

```
#include <omp.h>
#include <cstdio>
int main (int argc, char *argv[])
{
int th_id, nthreads;
#pragma omp parallel private(th_id) shared(nthreads)
{
th_id = omp_get_thread_num();
printf("Hello World from thread %d\n", th_id);
#pragma omp barrier
if ( th_id == 0 ) {
nthreads = omp_get_num_threads();
printf("There are %d threads\n",nthreads);
}
}
return 0;
}
```

Hello world, yet another variant

```
#include <cstdio>
#include <omp.h>
int main(int argc, char *argv[])
{
 omp_set_num_threads(4);
#pragma omp parallel
 {
   int id = omp_get_thread_num();
   int nproc = omp_get_num_threads();
   cout << "Hello world with id number and processes " <<  id <<  npro
 }
return 0;
}
```

Variables declared outside of the parallel region are shared by all threads If a variable like **id** is declared outside of the

```
#pragma omp parallel,
```

it would have been shared by various the threads, possibly causing erroneous output

- Why? What would go wrong? Why do we add possibly?

Important OpenMP library routines

- **int omp get num threads ()**, returns the number of threads inside a parallel region
- **int omp get thread num ()**, returns the a thread for each thread inside a parallel region
- **void omp set num threads (int)**, sets the number of threads to be used
- **void omp set nested (int)**, turns nested parallelism on/off

Private variables

Private clause can be used to make thread- private versions of such variables:

```
#pragma omp parallel private(id)
{
 int id = omp_get_thread_num();
 cout << "My thread num" << id << endl;
}
```

- What is their value on entry? Exit?
- OpenMP provides ways to control that
- Can use default(none) to require the sharing of each variable to be described

Master region

It is often useful to have only one thread execute some of the code in a parallel region. I/O statements are a common example

```
#pragma omp parallel
{
  #pragma omp master
   {
      int id = omp_get_thread_num();
      cout << "My thread num" << id << endl;
   }
}
```

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a for-loop:

```
#pragma omp for
```

- Clauses can be added, such as
  - **schedule(static, chunk size)**
  - **schedule(dynamic, chunk size)**
  - **schedule(guided, chunk size)** (non-deterministic allocation)
  - **schedule(runtime)**
  - **private(list of variables)**
  - **reduction(operator:variable)**
  - **nowait**

Parallel computations and loops

OpenMP provides an easy way to parallelize a loop

```
 #pragma omp parallel for
   for (i=0; i<n; i++) c[i] = a[i];
```

OpenMP handles index variable (no need to declare in for loop or make private)

Which thread does which values? Several options.

Scheduling of loop computations

We can let the OpenMP runtime decide. The decision is about how the loop iterates are scheduled and OpenMP defines three choices of loop scheduling:

1. Static: Predefined at compile time. Lowest overhead, predictable
2. Dynamic: Selection made at runtime
3. Guided: Special case of dynamic; attempts to reduce overhead

Example code for loop scheduling

```c
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
int main (int argc, char *argv[])
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk)
for (i=0; i < N; i++) c[i] = a[i] + b[i];
} /* end of parallel region */
}
```

Example code for loop scheduling, guided instead of dynamic

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
int main (int argc, char *argv[])
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
#pragma omp for schedule(guided,chunk)
for (i=0; i < N; i++) c[i] = a[i] + b[i];
} /* end of parallel region */
}
```

More on Parallel for loop

- The number of loop iterations cannot be non-deterministic; break, return, exit, goto not allowed inside the for-loop
- The loop index is private to each thread
- A reduction variable is special
  - During the for-loop there is a local private copy in each thread
  - At the end of the for-loop, all the local copies are combined together by the reduction operation
- Unless the nowait clause is used, an implicit barrier synchronization will be added at the end by the compiler

```
// #pragma omp parallel and #pragma omp for
```

can be combined into

```
#pragma omp parallel for
```

What can happen with this loop?

What happens with code like this

```
#pragma omp parallel for
for (i=0; i<n; i++) sum += a[i]*a[i];
```

All threads can access the **sum** variable, but the addition is not atomic! It is important to avoid race between threads. So-called reductions in OpenMP are thus important for performance and for obtaining correct results. OpenMP lets us indicate that a variable is used for a reduction with a particular operator. The above code becomes

```
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<n; i++) sum += a[i]*a[i];
```

Inner product

$$\sum_{i=0}^{n-1} a_i b_i$$

```
int i;
double sum = 0.;
/* allocating and initializing arrays */
/* ... */
#pragma omp parallel for default(shared) private(i) reduction(+:sum)
 for (i=0; i<N; i++) sum += a[i]*b[i];
}
```

Different threads do different tasks

Different threads do different tasks independently, each section is executed by one thread.

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
funcA ();
#pragma omp section
funcB ();
#pragma omp section
funcC ();
}
}
```

Single execution

```
#pragma omp single { ... }
```

The code is executed by one thread only, no guarantee which thread
Can introduce an implicit barrier at the end

```
#pragma omp master { ... }
```

Code executed by the master thread, guaranteed and no implicit barrier at the end.

Coordination and synchronization

```
#pragma omp barrier
```

Synchronization, must be encountered by all threads in a team (or none)

```
#pragma omp ordered { a block of codes }
```

is another form of synchronization (in sequential order). The form

```
#pragma omp critical { a block of codes }
```

and

```
#pragma omp atomic { single assignment statement }
```

is more efficient than

```
#pragma omp critical
```

Data scope

- OpenMP data scope attribute clauses:
  - **shared**
  - **private**
  - **firstprivate**
  - **lastprivate**
  - **reduction**

What are the purposes of these attributes

- define how and which variables are transferred to a parallel region (and back)
- define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the **private** clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It is the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The **firstprivate** clause combines the behavior of the private clause with automatic initialization.
- The **lastprivate** clause combines the behavior of the private clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Parallelizing nested for-loops

- Serial code

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        a[i][j] = b[i][j] + c[i][j];
    }
}
```

- Parallelization

```
#pragma omp parallel for private(j)
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
       a[i][j] = b[i][j] + c[i][j];
    }
}
```

- Why not parallelize the inner loop? to save overhead of repeated thread forks-joins
- Why must **j** be private? To avoid race condition among the threads

Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.

```
#pragma omp parallel num_threads(4)
{
/* .... */
#pragma omp parallel num_threads(2)
{
//
}
}
```

Parallel tasks

```
#pragma omp task
#pragma omp parallel shared(p_vec) private(i)
{
#pragma omp single
{
for (i=0; i<N; i++) {
  double r = random_number();
  if (p_vec[i] > r) {
#pragma omp task
   do_work (p_vec[i]);
```

Common mistakes

Race condition

```
int nthreads;
#pragma omp parallel shared(nthreads)
{
nthreads = omp_get_num_threads();
}
```

Deadlock

```
#pragma omp parallel
{
...
#pragma omp critical
{
...
#pragma omp barrier
}
}
```

Not all computations are simple

Not all computations are simple loops where the data can be evenly divided among threads without any dependencies between threads

An example is finding the location and value of the largest element in an array

```
for (i=0; i<n; i++) {
   if (x[i] > maxval) {
      maxval = x[i];
      maxloc = i;
   }
}
```

Not all computations are simple, competing threads

All threads are potentially accessing and changing the same values, **maxloc** and **maxval**.

1. OpenMP provides several ways to coordinate access to shared values

```
#pragma omp atomic
```

1. Only one thread at a time can execute the following statement (not block). We can use the critical option

```
#pragma omp critical
```

1. Only one thread at a time can execute the following block

Atomic may be faster than critical but depends on hardware

How to find the max value using OpenMP

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as

```
#pragma omp parallel for
 for (i=0; i<n; i++) {
    if (x[i] > maxval) {
      maxval = x[i];
      maxloc = i;
    }
}
```

Then deal with the race conditions

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as

```
#pragma omp parallel for
 for (i=0; i<n; i++) {
#pragma omp critical
  {
     if (x[i] > maxval) {
       maxval = x[i];
       maxloc = i;
     }
  }
}
```

Exercise: write a code which implements this and give an estimate on performance. Perform several runs, with a serial code only with and without vectorization and compare the serial code with the one that uses OpenMP. Run on different archictectures if you can.

What can slow down OpenMP performance?

Give it a thought!

What can slow down OpenMP performance?

Performance poor because we insisted on keeping track of the maxval and location during the execution of the loop.

- We do not care about the value during the execution of the loop, just the value at the end.

This is a common source of performance issues, namely the description of the method used to compute a value imposes additional, unnecessary requirements or properties

Idea: Have each thread find the maxloc in its own data, then combine and use temporary arrays indexed by thread number to hold the values found by each thread

Find the max location for each thread

```
int maxloc[MAX_THREADS], mloc;
double maxval[MAX_THREADS], mval;
#pragma omp parallel shared(maxval,maxloc)
{
  int id = omp_get_thread_num();
  maxval[id] = -1.0e30;
#pragma omp for
   for (int i=0; i<n; i++) {
       if (x[i] > maxval[id]) {
           maxloc[id] = i;
           maxval[id] = x[i];
       }
    }
}
```

Combine the values from each thread

```
#pragma omp flush (maxloc,maxval)
#pragma omp master
  {
    int nt = omp_get_num_threads();
    mloc = maxloc[0];
    mval = maxval[0];
    for (int i=1; i<nt; i++) {
        if (maxval[i] > mval) {
           mval = maxval[i];
           mloc = maxloc[i];
        }
     }
   }
```

Note that we let the master process perform the last operation.

Matrix-matrix multiplication

This code computes the norm of a vector using OpenMp

```
//  OpenMP program to compute vector norm by adding two other vectors
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include  <omp.h>
# include <ctime>

using namespace std; // note use of namespace
int main (int argc, char* argv[])
{
  // read in dimension of vector
  int n = atoi(argv[1]);
  double *a, *b, *c;
  int i;
  int thread_num;
  double wtime, Norm2, s, angle;
  cout << "  Perform addition of two vectors and compute the norm-2."
  omp_set_num_threads(4);
  thread_num = omp_get_max_threads ();
  cout << "  The number of processors available = " << omp_get_num_pro
  cout << "  The number of threads available    = " << thread_num <<
  cout << "  The matrix order n                 = " << n << endl;

  s = 1.0/sqrt( (double) n);
  wtime = omp_get_wtime ( );
  // Allocate space for the vectors to be used
```

Matrix-matrix multiplication

This the matrix-matrix multiplication code with plain c++ memory allocation using OpenMP

```
//  Matrix-matrix multiplication and Frobenius norm of a matrix with O
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include  <omp.h>
# include <ctime>

using namespace std; // note use of namespace
int main (int argc, char* argv[])
{
  // read in dimension of square matrix
  int n = atoi(argv[1]);
  double **A, **B, **C;
  int i, j, k;
  int thread_num;
  double wtime, Fsum, s, angle;
  cout << "  Compute matrix product C = A * B and Frobenius norm." <<
  omp_set_num_threads(4);
  thread_num = omp_get_max_threads ();
  cout << "  The number of processors available = " << omp_get_num_pro
  cout << "  The number of threads available    = " << thread_num <<
  cout << "  The matrix order n                 = " << n << endl;

  s = 1.0/sqrt( (double) n);
```