How to parallelize a Variational Monte Carlo code with MPI and OpenMP

Morten Hjorth-Jensen[1,2]

National Superconducting Cyclotron Laboratory and Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, USA[1]

Department of Physics, University of Oslo, Oslo, Norway[2]

Spring 2015

Your background

- You have some experience in programming but have never tried to parallelize your codes
- Here I will base my examples on C/C++ using Message Passing Interface (MPI) and OpenMP.
- I will also give you some simple hints on how to run and install codes on your laptop/office PC
- The programs and slides can be found at the weblink
- Good text: Karniadakis and Kirby, Parallel Scientific Computing in C++ and MPI, Cambridge.

We will discuss Message passing interface (MPI) and OpenMP.

===== =====

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop or PC. You can install MPICH2 on your laptop/PC.
- Test by typing *which mpd*
- When you are convinced that your codes run correctly, you start your production runs on available supercomputers, in our case *smaug* locally (to be discussed after Easter).

How do I run MPI on a PC/Laptop?

Most machines at computer labs at UiO are quad-cores

- Compile with mpicxx or mpic++ or mpif90
- Set up collaboration between processes and run

```
mpd --ncpus=4 &
#  run code with
mpiexec -n 4 ./nameofprog
```

Here we declare that we will use 4 processes via the *-ncpus* option and via $-n4$ when running.

- End with *mpdallexit*

Can I do it on my own PC/laptop?

Of course:

- go to the website of Argonne National Lab
- follow the instructions and install it on your own PC/laptop
- Versions for Ubuntu/Linux, windows and mac
- For windows, you may think of installing WUBI

What is Message Passing Interface (MPI)?

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C/C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.
MPI programs should be able to run on all possible machines and run all MPI implementetations without change.
An MPI computation is a collection of processes communicating with messages.

Going Parallel with MPI

Task parallelism: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations or numerical integration are examples of this.
MPI is a message-passing library where all the routines have corresponding C/C++-binding

```
MPI_Command_name
```

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

```
MPI_COMMAND_NAME
```

MPI is a library

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use.
Designed for high performance.
Insert communication and synchronization functions where necessary.

The basic ideas of parallel computing

- Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.
- Multiple processors are involved to solve a global problem.
- The essence is to divide the entire computation evenly among collaborative processors. Divide and conquer.

A rough classification of hardware models

- Conventional single-processor computers can be called SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, which use a large number of processing units to execute the same instruction on different data.
- Modern parallel computers are so-called MIMD (multiple-instruction-multiple-data) machines and can execute different instruction streams in parallel on different data.

Shared memory and distributed memory

- One way of categorizing modern parallel computers is to look at the memory configuration.
- In shared memory systems the CPUs share the same address space. Any CPU can access any data in the global memory.
- In distributed memory systems each CPU has its own memory.

The CPUs are connected by some network and may exchange messages.

Different parallel programming paradigms

- **Task parallelism**: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation is one example. Integration is another. However this paradigm is of limited use.
- **Data parallelism**: use of multiple threads (e.g. one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between processors are often hidden, thus easy to program. However, the user surrenders much control to a specialized compiler. Examples of data parallelism are compiler-based parallelization and OpenMP directives.

Different parallel programming paradigms

- **Message passing**: all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.
- This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult.

SPMD (single-program-multiple-data)

Although message-passing programming supports MIMD, it suffices with an SPMD (single-program-multiple-data) model, which is flexible enough for practical cases:

- Same executable for all the processors.
- Each processor works primarily with its assigned local data.
- Progression of code is allowed to differ between synchronization points.
- Possible to have a master/slave model. The standard option in Monte Carlo calculations and numerical integration.

Today's situation of parallel computing

- Distributed memory is the dominant hardware configuration. There is a large diversity in these machines, from MPP (massively parallel processing) systems to clusters of off-the-shelf PCs, which are very cost-effective.
- Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware. It is primarily used for the distributed memory systems, but can also be used on shared memory systems.
- Modern nodes have nowadays several cores, which makes it interesting to use both shared memory (the given node) and distributed memory (several nodes with communication). This leads often to codes which use both MPI and OpenMP.

Our lectures will focus on both MPI and OpenMP.

Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization**
- **Overhead of communication**
- **Extra computation due to parallelization**

Due to the above overhead and that certain part of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among P processors.

Bindings to MPI routines

MPI is a message-passing library where all the routines have corresponding C/C++-binding

```
MPI_Command_name
```

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

```
MPI_COMMAND_NAME
```

The discussion in these slides focuses on the C++ binding.

Communicator

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default communicator *MPI_COMM_WORLD* contains all the MPI processes.
- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

Some of the most important MPI functions

- *MPI_Init* - initiate an MPI computation
- *MPI_Finalize* - terminate the MPI computation and clean up
- *MPI_Comm_size* - how many processes participate in a given MPI communicator?
- *MPI_Comm_rank* - which one am I? (A number between 0 and size-1.)
- *MPI_Send* - send a message to a particular process within an MPI communicator
- *MPI_Recv* - receive a message from a particular process within an MPI communicator
- *MPI_reduce* or *MPI_Allreduce*, send and receive messages

The first MPI C/C++ program

Let every process write "Hello world" (oh not this program again!!) on the standard output.

```
using namespace std;
#include <mpi.h>
#include <iostream>
int main (int nargs, char* args[])
{
int numprocs, my_rank;
//   MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
cout << "Hello world, I have  rank " << my_rank << " out of "
     << numprocs << endl;
//  End MPI
MPI_Finalize ();
```

The Fortran program

```
PROGRAM hello
INCLUDE "mpif.h"
INTEGER:: size, my_rank, ierr

CALL  MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
WRITE(*,*)"Hello world, I've rank ",my_rank," out of ",size
CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

Note 1

- The output to screen is not ordered since all processes are trying to write to screen simultaneously.
- It is the operating system which opts for an ordering.
- If we wish to have an organized output, starting from the first process, we may rewrite our program as in the next example.

Ordered output with *MPI_Barrier*

```
int main (int nargs, char* args[])
{
 int numprocs, my_rank, i;
 MPI_Init (&nargs, &args);
 MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
 for (i = 0; i < numprocs; i++) {}
 MPI_Barrier (MPI_COMM_WORLD);
 if (i == my_rank) {
 cout << "Hello world, I have  rank " << my_rank <<
        " out of " << numprocs << endl;}
      MPI_Finalize ();
```

Note 2

- Here we have used the *MPI_Barrier* function to ensure that that every process has completed its set of instructions in a particular order.
- A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here *MPI_COMM_WORLD*) have called *MPI_Barrier*.
- The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like *MPI_ALLREDUCE* to be discussed later, have the same property; that is, no process can exit the operation until all processes have started.

However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

Ordered output with *MPI_Recv* and *MPI_Send*

```
.....
int numprocs, my_rank, flag;
MPI_Status status;
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
if (my_rank > 0)
MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100,
          MPI_COMM_WORLD, &status);
cout << "Hello world, I have  rank " << my_rank << " out of "
<< numprocs << endl;
if (my_rank < numprocs-1)
MPI_Send (&my_rank, 1, MPI_INT, my_rank+1,
          100, MPI_COMM_WORLD);
MPI_Finalize ();
```

Note 3

The basic sending of messages is given by the function *MPI_SEND*, which in C/C++ is defined as

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)}
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

Note 4

Once you have sent a message, you must receive it on another task. The function *MPI_RECV* is similar to the send call.

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int source,
              int tag, MPI_Comm comm, MPI_Status *status )
```

The arguments that are different from those in MPI_SEND are **buf** which is the name of the variable where you will be storing the received data, **source** which replaces the destination in the send command. This is the return ID of the sender.
Finally, we have used *MPI_Status_status*, where one can check if the receive was completed.
The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Numerical integration in parallel

Integrating π

- The code example computes π using the trapezoidal rules.
- The trapezoidal rule

$$I = \int_a^b f(x)dx \approx h(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f($$

Dissection of trapezoidal rule with *MPI_reduce*

```
//    Trapezoidal rule and numerical integration usign MPI
using namespace std;
#include <mpi.h>
#include <iostream>

//     Here we define various functions called by the main program

double int_function(double );
double trapezoidal_rule(double , double , int , double (*)(double));

//   Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum, local_sum;
  double  time_start, time_end, total_time;
```

Dissection of trapezoidal rule with *MPI_reduce*

```
  //  MPI initializations
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  time_start = MPI_Wtime();
  //  Fixed values for a, b and n
  a = 0.0 ; b = 1.0;  n = 1000;
  h = (b-a)/n;    // h is the same for all processes
  local_n = n/numprocs;
  // make sure n > numprocs, else integer division gives zero
  // Length of each process' interval of
  // integration = local_n*h.
  local_a = a + my_rank*local_n*h;
  local_b = local_a + local_n*h;
```

Integrating with **MPI**

```
  total_sum = 0.0;
  local_sum = trapezoidal_rule(local_a, local_b, local_n,
                               &int_function);
  MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  if ( my_rank == 0) {
    cout << "Trapezoidal rule = " <<  total_sum << endl;
    cout << "Time = " <<  total_time
         << " on number of processors: "  << numprocs  << endl;
  }
  // End MPI
  MPI_Finalize ();
  return 0;
}  // end of main program
```

How do I use *MPI_reduce*?

Here we have used

```
MPI_reduce( void *senddata, void* resultdata, int count,
     MPI_Datatype datatype, MPI_Op, int root, MPI_Comm comm)
```

The two variables *senddata* and *resultdata* are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable *count* represents the total dimensionality, 1 in case of just one variable, while *MPI_Datatype* defines the type of variable which is sent and received.

The new feature is *MPI_Op*. It defines the type of operation we want to do.

More on *MPI_Reduce*

In our case, since we are summing the rectangle contributions from every process we define *MPI_Op* = *MPI_SUM*. If we have an array or matrix we can search for the largest og smallest element by sending either *MPI_MAX* or *MPI_MIN*. If we want the location as well (which array element) we simply transfer *MPI_MAXLOC* or *MPI_MINOC*. If we want the product we write *MPI_PROD*.

MPI_Allreduce is defined as

```
MPI_Allreduce( void *senddata, void* resultdata, int count,
        MPI_Datatype datatype, MPI_Op, MPI_Comm comm)
```

Dissection of trapezoidal rule with *MPI_reduce*

We use *MPI_reduce* to collect data from each process. Note also the use of the function *MPI_Wtime*.

```
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

Dissection of trapezoidal rule with *MPI_reduce*

```
//  this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int    j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
}  // end trapezoidal_rule
```

Optimization and profiling

Till now we have not paid much attention to speed and possible optimization possibilities inherent in the various compilers. We have compiled and linked as

```
mpic++  -c  mycode.cpp
mpic++  -o  mycode.exe  mycode.o
```

For Fortran replace with mpif90. This is what we call a flat compiler option and should be used when we develop the code. It produces normally a very large and slow code when translated to machine instructions. We use this option for debugging and for establishing the correct program output because every operation is done precisely as the user specified it.

It is instructive to look up the compiler manual for further instructions

```
man mpic++  >  out_to_file
```

More on optimization

We have additional compiler options for optimization. These may include procedure inlining where performance may be improved, moving constants inside loops outside the loop, identify potential parallelism, include automatic vectorization or replace a division with a reciprocal and a multiplication if this speeds up the code.

```
mpic++  -O3 -c  mycode.cpp
mpic++  -O3 -o  mycode.exe  mycode.o
```

This is the recommended option. **But you must check that you get the same results as previously**.

Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with

```
mpic++  -pg -O3 -c  mycode.cpp
mpic++  -pg -O3 -o  mycode.exe  mycode.o
```

After you have run the code you can obtain the profiling information via

```
gprof mycode.exe >  out_to_profile
```

When you have profiled properly your code, you must take out this option as it increases your CPU expenditure. For memory tests use "valgrind":"http:www.valgrind.org". An excellent GUI is also Qt, with debugging facilities.

Optimization and profiling

Other hints

- avoid if tests or call to functions inside loops, if possible.
- avoid multiplication with constants inside loops if possible

Bad code

```
for i = 1:n
    a(i) = b(i) +c*d
    e = g(k)
end
```

Better code

```
temp = c*d
for i = 1:n
    a(i) = b(i) + temp
end
e = g(k)
```

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

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

General code structure

```
#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 <stdio.h>
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;
}
```

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

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**

Example code

```
#include <omp.h>
#define CHUNKSIZE 100
#define N
1000
main ()
{
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 */
}
```

More on Parallel for loop

- The number of loop iterations can not 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
```

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
}
}
```

Matrix-matrix multiplication

```
# include <cstdlib>
# include <iostream>
# include <cmath>
# include <ctime>
# include <omp.h>

using namespace std;

// Main function
int main ( )
{
// brute force coding of arrays
  double a[500][500];
  double angle;
  double b[500][500];
  double c[500][500];
  int i;
  int j;
  int k;
```

Matrix-matrix multiplication

```
  int n = 500;
  double pi = acos(-1.0);
  double s;
  int thread_num;
  double wtime;

  cout << "\n";
  cout << "  C++/OpenMP version\n";
  cout << "  Compute matrix product C = A * B.\n";

  thread_num = omp_get_max_threads ( );

//
//  Loop 1: Evaluate A.
//
  s = 1.0 / sqrt ( ( double ) ( n ) );

  wtime = omp_get_wtime ( );
```

Matrix-matrix multiplication

```
# pragma omp parallel shared ( a, b, c, n, pi, s )
private ( angle, i, j, k )
{
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      angle = 2.0 * pi * i * j / ( double ) n;
      a[i][j] = s * ( sin ( angle ) + cos ( angle ) );
    }
  }
//
//  Loop 2: Copy A into B.
//
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      b[i][j] = a[i][j];
    }
  }
```

Matrix-matrix multiplication

```
//  Loop 3: Compute C = A * B.
//
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      c[i][j] = 0.0;
      for ( k = 0; k < n; k++ )
      {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
  wtime = omp_get_wtime ( ) - wtime;
  cout << "  Elapsed seconds = " << wtime << "\n";
  cout << "  C(100,100)  = " << c[99][99] << "\n";
//
//  Terminate.
//
  cout << "\n";
  cout << "  Normal end of execution.\n";
  return 0;
```