Computational Physics Lectures: How to optimize codes, from vectorization to parallelization

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

2016

Content

- Simple compiler options
- Tools to benchmark your code
- Machine architectures
- What is vectorization?
- How to measure code performance
- Parallelization with OpenMP
- Parallelization with MPI
- Vectorization and parallelization, examples

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

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

For Fortran replace with for example **gfortran** or **ifort**. 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 by writing

```
man c++
```

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.

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

This (other options are -O2 or -Ofast) is the recommended option.

Optimization and profiling

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

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

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

```
gprof mycode.exe >  ProfileOutput
```

When you have profiled properly your code, you must take out this option as it slows down performance. For memory tests use valgrind. An excellent environment for all these aspects, and much more, is Qt creator.

Optimization and debugging

Adding debugging options is a very useful alternative under the development stage of a program. You would then compile with

```
c++  -g -O0 -c  mycode.cpp
c++  -g -O0 -o  mycode.exe  mycode.o
```

This option generates debugging information allowing you to trace for example if an array is properly allocated. Some compilers work best with the no optimization option **-O0**.

Other optimization flags. Depending on the compiler, one can add flags which generate code that catches integer overflow errors. The flag **-ftrapv** does this for the CLANG compiler on OS X operating systems.

Other hints

In general, irrespective of compiler options, it is useful to

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

Here is an example of a part of a program where specific operations lead to a slower code

```
k = n-1;
for (i = 0; i < n; i++){
    a[i] = b[i] +c*d;
    e = g[k];
}
```

A better code is

```
temp = c*d;
for (i = 0; i < n; i++){
    a[i] = b[i] + temp;
}
e = g[n-1];
```

Here we avoid a repeated multiplication inside a loop. Most compilers, depending on compiler flags, identify and optimize such bottlenecks on their own, without requiring any particular action by the programmer. However, it is always useful to single out and avoid code examples like the first one discussed here.

Vectorization and the basic idea behind parallel computing

Present CPUs are highly parallel processors with varying levels of parallelism. The typical situation can be described via the following three statements.

- 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.

Before we proceed with a more detailed discussion of topics like vectorization and parallelization, we need to remind ourselves about some basic features of different hardware models.

A rough classification of hardware models

- Conventional single-processor computers are named SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, using 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 simulations represent a typical situation. Integration is another. However this paradigm is of limited use.
- **Data parallelism**: use of multiple threads (e.g. one or more threads per processor) to dissect loops over arrays etc. 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.

What is vectorization?

Vectorization is a special case of **Single Instructions Multiple Data** (SIMD) to denote a single instruction stream capable of operating on multiple data elements in parallel. We can think of vectorization as the unrolling of loops accompanied with SIMD instructions.

Vectorization is the process of converting an algorithm that performs scalar operations (typically one operation at the time) to vector operations where a single operation can refer to many simultaneous operations. Consider the following example

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

If the code is not vectorized, the compiler will simply start with the first element and then perform subsequent additions operating on one address in memory at the time.

Number of elements that can acted upon

A SIMD instruction can operate on multiple data elements in one single instruction. It uses the so-called 128-bit SIMD floating-point register. In this sense, vectorization adds some form of parallelism since one instruction is applied to many parts of say a vector.

The number of elements which can be operated on in parallel range from four single-precision floating point data elements in so-called Streaming SIMD Extensions and two double-precision floating-point data elements in Streaming SIMD Extensions 2 to sixteen byte operations in a 128-bit register in Streaming SIMD Extensions 2. Thus, vector-length ranges from 2 to 16, depending on the instruction extensions used and on the data type.

IN summary, our instructions operate on 128 bit (16 byte) operands

- 4 floats or ints
- 2 doubles
- Data paths 128 bits vide for vector unit

Number of elements that can acted upon, examples

We start with the simple scalar operations given by

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

If the code is not vectorized and we have a 128-bit register to store a 32 bits floating point number, it means that we have 3×32 bits that are not used. For the first element we have

0	1	2	3
a[0]=	not used	not used	not used
b[0]+	not used	not used	not used
c[0]	not used	not used	not used

We have thus unused space in our SIMD registers. These registers could hold three additional integers.

Operation counts for scalar operation

The code

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

has for n repeats

1. one load for $a[i]$ in address 1
2. one load for $b[i]$ in address 2
3. add $a[i]$ and $b[i]$ to give $c[i]$
4. store $c[i]$ in address 2

Number of elements that can acted upon, examples

If we vectorize the code, we can perform, with a 128-bit register four simultaneous operations, that is we have

```
for (i = 0; i < n; i+=4){
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
    a[i+3] = b[i+3] + c[i+3];
}
```

displayed here as

0	1	2	3
a[0]=	a[1]=	a[2]=	a[3]=
b[0]+	b[1]+	b[2]+	b[3]+
c[0]	c[1]	c[2]	c[3]

Four additions are now done in a single step.

Number of operations when vectorized

For $n/4$ repeats assuming floats or integers

1. one vector load for $a[i]$ in address 1
2. one load for $b[i]$ in address 2
3. add $a[i]$ and $b[i]$ to give $c[i]$
4. store $c[i]$ in address 2

A simple test case with and without vectorization

We implement these operations in a simple c++ program as

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include "time.h"

using namespace std; // note use of namespace
int main (int argc, char* argv[])
{
  int i = atoi(argv[1]);
  double *a, *b, *c;
  a = new double[i];
  b = new double[i];
  c = new double[i];
  for (int j = 0; j < i; j++) {
    a[j] = 0.0;
    b[j] = cos(j*1.0);
    c[j] = sin(j*3.0);
  }
  clock_t start, finish;
  start = clock();
  for (int j = 0; j < i; j++) {
    a[j] = b[j]+b[j]*c[j];
  }
  finish = clock();
  double timeused = (double) (finish - start)/(CLOCKS_PER_SEC );
  cout << setiosflags(ios::showpoint | ios::uppercase);
  cout << setprecision(10) << setw(20) << "Time used  for vector addition and multiplication=" << time
  delete [] a;
  delete [] b;
  delete [] c;
  return 0;
}
```

Compiling with and without vectorization

We can compile and link without vectorization

```
c++ -o novec.x vecexample.cpp
```

and with vectorization (and additional optimizations)

```
c++ -O3 -o  vec.x vecexample.cpp
```

The speedup depends on the size of the vectors. In the example here we have run with 10^7 elements. The example here was run on a PC with ubuntu 14.04 as operating system and an Intel i7-4790 CPU running at 3.60 GHz.

```
Compphys:~ hjensen$ ./vec.x 10000000
Time used  for vector addition = 0.0100000
Compphys:~ hjensen$ ./novec.x 10000000
Time used  for vector addition = 0.03000000000
```

This particular C++ compiler speeds up the above loop operations with a factor of 3. Performing the same operations for 10^8 elements results only in a factor 1.4. The result will however vary from compiler to compiler. In general however, with optimization flags like $-O3$ or $-Ofast$, we gain a considerable speedup if our code can be vectorized. Many of these operations can be done automatically by your compiler. These automatic or near automatic compiler techniques improve performance considerably. Below we will see an example on a different behavior if we use the **clang** compiler.

Automatic vectorization and vectorization inhibitors, criteria

Not all loops can be vectorized, as discussed in Intel's guide to vectorization

An important criteria is that the loop counter n is known at the entry of the loop.

```
for (int j = 0; j < n; j++) {
  a[j] = cos(j*1.0);
}
```

The variable n does need to be known at compile time. However, this variable must stay the same for the entire duration of the loop. It implies that an exit statement inside the loop cannot be data dependent.

Automatic vectorization and vectorization inhibitors, exit criteria

An exit statement should in general be avoided. If the exit statement contains data-dependent conditions, the loop cannot be vectorized. The following is an example of a non-vectorizable loop

```
for (int j = 0; j < n; j++) {
  a[j] = cos(j*1.0);
  if (a[j] < 0 ) break;
}
```

Avoid loop termination conditions and opt for a single entry loop variable n. The lower and upper bounds have to be kept fixed within the loop.

Automatic vectorization and vectorization inhibitors, straight-line code

SIMD instructions perform the same type of operations multiple times. A **switch** statement leads thus to a non-vectorizable loop since different statemens cannot branch. The following code can however be vectorized since the **if** statement is implemented as a masked assignment.

```
for (int j = 0; j < n; j++) {
  double x  = cos(j*1.0);
  if (x > 0 ) {
     a[j] =  x*sin(j*2.0);
  }
  else {
     a[j] = 0.0;
  }
}
```

These operations can be performed for all data elements but only those elements which the mask evaluates as true are stored. In general, one should avoid branches such as **switch**, **go to**, or **return** statements or **if** constructs that cannot be treated as masked assignments.

Automatic vectorization and vectorization inhibitors, nested loops

Only the innermost loop of the following example is vectorized

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
         a[i][j] += b[i][j];
    }
}
```

The exception is if an original outer loop is transformed into an inner loop as the result of compiler optimizations.

Automatic vectorization and vectorization inhibitors, function calls

Calls to programmer defined functions ruin vectorization. However, calls to intrinsic functions like $\sin x$, $\cos x$, $\exp x$ etc are allowed since they are normally efficiently vectorized. The following example is fully vectorizable

```
for (int i = 0; i < n; i++) {
    a[i] = log10(i)*cos(i);
}
```

Similarly, **inline** functions defined by the programmer, allow for vectorization since the function statements are glued into the actual place where the function is called.

Automatic vectorization and vectorization inhibitors, data dependencies

One has to keep in mind that vectorization changes the order of operations inside a loop. A so-called read-after-write statement with an explicit flow dependency cannot be vectorized. The following code

```
double b = 15.;
for (int i = 1; i < n; i++) {
    a[i] = a[i-1] + b;
}
```

is an example of flow dependency and results in wrong numerical results if vectorized. For a scalar operation, the value $a[i-1]$ computed during the iteration is loaded into the right-hand side and the results are fine. In vector mode however, with a vector length of four, the values $a[0]$, $a[1]$, $a[2]$ and $a[3]$ from the previous loop will be loaded into the right-hand side and produce wrong results. That is, we have

```
a[1] = a[0] + b;
a[2] = a[1] + b;
a[3] = a[2] + b;
a[4] = a[3] + b;
```

and if the two first iterations are executed at the same by the SIMD instruction, the value of say $a[1]$ could be used by the second iteration before it has been calculated by the first iteration, leading thereby to wrong results.

Automatic vectorization and vectorization inhibitors, more data dependencies

On the other hand, a so-called write-after-read statement can be vectorized. The following code

```
double b = 15.;
for (int i = 1; i < n; i++) {
    a[i-1] = a[i] + b;
}
```

is an example of flow dependency that can be vectorized since no iteration with a higher value of i can complete before an iteration with a lower value of i. However, such code leads to problems with parallelization.

Automatic vectorization and vectorization inhibitors, memory stride

For C++ programmers it is also worth keeping in mind that an array notation is preferred to the more compact use of pointers to access array elements. The compiler can often not tell if it is safe to vectorize the code.

When dealing with arrays, you should also avoid memory stride, since this slows down considerably vectorization. When you access array element, write for

example the inner loop to vectorize using unit stride, that is, access successively the next array element in memory, as shown here

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
         a[i][j] += b[i][j];
    }
}
```

Memory management

The main memory contains the program data

- Cache memory contains a copy of the main memory data
- Cache is faster but consumes more space and power. It is normally assumed to be much faster than main memory
- Registers contain working data only
  - Modern CPUs perform most or all operations only on data in register
- Multiple Cache memories contain a copy of the main memory data
  - Cache items accessed by their address in main memory
  - L1 cache is the fastest but has the least capacity
  - L2, L3 provide intermediate performance/size tradeoffs

Loads and stores to memory can be as important as floating point operations when we measure performance.

Memory and communication

- Most communication in a computer is carried out in chunks, blocks of bytes of data that move together
- In the memory hierarchy, data moves between memory and cache, and between different levels of cache, in groups called lines
  - Lines are typically 64-128 bytes, or 8-16 double precision words
  - Even if you do not use the data, it is moved and occupies space in the cache
- This performance feature is not captured in most programming languages

Measuring performance

How do we measure erformance? What is wrong with this code to time a loop?

```
clock_t start, finish;
start = clock();
for (int j = 0; j < i; j++) {
  a[j] = b[j]+b[j]*c[j];
}
finish = clock();
double timeused = (double) (finish - start)/(CLOCKS_PER_SEC );
```

Problems with measuring time

1. Timers are not infinitely accurate
2. All clocks have a granularity, the minimum time that they can measure
3. The error in a time measurement, even if everything is perfect, may be the size of this granularity (sometimes called a clock tick)
4. Always know what your clock granularity is
5. Ensure that your measurement is for a long enough duration (say 100 times the **tick**)

Problems with cold start

What happens when the code is executed? The assumption is that the code is ready to execute. But

1. Code may still be on disk, and not even read into memory.
2. Data may be in slow memory rather than fast (which may be wrong or right for what you are measuring)
3. Multiple tests often necessary to ensure that cold start effects are not present
4. Special effort often required to ensure data in the intended part of the memory hierarchy.

Problems with smart compilers

1. If the result of the computation is not used, the compiler may eliminate the code
2. Performance will look impossibly fantastic
3. Even worse, eliminate some of the code so the performance looks plausible
4. Ensure that the results are (or may be) used.

Problems with interference

1. Other activities are sharing your processor
   - Operating system, system demons, other users
     - Some parts of the hardware do not always perform with exactly the same performance
2. Make multiple tests and report
3. Easy choices include
   - Average tests represent what users might observe over time

Problems with measuring performance

1. Accurate, reproducible performance measurement is hard
2. Think carefully about your experiment:
3. What is it, precisely, that you want to measure
4. How representative is your test to the situation that you are trying to measure?

Thomas algorithm for tridiagonal linear algebra equations

$$\begin{pmatrix} b_0 & c_0 & & & \\ a_0 & b_1 & c_1 & & \\ & & \ddots & & \\ & & a_{m-3} & b_{m-2} & c_{m-2} \\ & & & a_{m-2} & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{m-2} \\ f_{m-1} \end{pmatrix}$$

Thomas algorithm, forward substitution

The first step is to multiply the first row by a_0/b_0 and subtract it from the second row. This is known as the forward substitution step. We obtain then

$$a_i = 0,$$

$$b_i = b_i - \frac{a_{i-1}}{b_{i-1}} c_{i-1},$$

and

$$f_i = f_i - \frac{a_{i-1}}{b_{i-1}} f_{i-1}.$$

At this point the simplified equation, with only an upper triangular matrix takes the form

$$\begin{pmatrix} b_0 & c_0 & & & \\ & b_1 & c_1 & & \\ & & \ddots & & \\ & & & b_{m-2} & c_{m-2} \\ & & & & b_{m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-2} \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{m-2} \\ f_{m-1} \end{pmatrix}$$

Thomas algorithm, backward substitution

The next step is the backward substitution step. The last row is multiplied by c_{N-3}/b_{N-2} and subtracted from the second to last row, thus eliminating c_{N-3} from the last row. The general backward substitution procedure is

$$c_i = 0,$$

and

$$f_{i-1} = f_{i-1} - \frac{c_{i-1}}{b_i} f_i$$

All that ramains to be computed is the solution, which is the very straight forward process of

$$x_i = \frac{f_i}{b_i}$$

Thomas algorithm and counting of operations (floating point and memory)

Operation	Floating Point
Memory Reads	$14(N-2)$
Memory Writes	$4(N-2)$
Subtractions	$3(N-2)$
Multiplications	$3(N-2)$
Divisions	$4(N-2)$

```
// Forward substitution
// Note that we can simplify by precalculating a[i-1]/b[i-1]
  for (int i=1; i < n; i++) {
     b[i] = b[i] - (a[i-1]*c[i-1])/b[i-1];
     f[i] = g[i] - (a[i-1]*f[i-1])/b[i-1];
  }
  x[n-1] = f[n-1] / b[n-1];
  // Backwards substitution
  for (int i = n-2; i >= 0; i--) {
     f[i] = f[i] - c[i]*f[i+1]/b[i+1];
     x[i] = f[i]/b[i];
  }
```

Example: Transpose of a matrix

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include "time.h"

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;
  // Allocate space for the two matrices
  A = new double*[n]; B = new double*[n];
  for (int i = 0; i < n; i++){
    A[i] = new double[n];
    B[i] = new double[n];
  }
  // Set up values for matrix A
  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++) {
      A[i][j] =  cos(i*1.0)*sin(j*3.0);
    }
  }
  clock_t start, finish;
  start = clock();
  // Then compute the transpose
  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++) {
      B[i][j]= A[j][i];
    }
  }

  finish = clock();
  double timeused = (double) (finish - start)/(CLOCKS_PER_SEC );
  cout << setiosflags(ios::showpoint | ios::uppercase);
  cout << setprecision(10) << setw(20) << "Time used  for setting up transpose of matrix=" << timeused

  // Free up space
  for (int i = 0; i < n; i++){
    delete[] A[i];
    delete[] B[i];
  }
  delete[] A;
  delete[] B;
  return 0;
}
```

Matrix-matrix multiplication

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

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include <iomanip>
#include "time.h"

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;
  // Allocate space for the three matrices
  A = new double*[n]; B = new double*[n]; C = new double*[n];
  for (int i = 0; i < n; i++){
    A[i] = new double[n];
    B[i] = new double[n];
    C[i] = new double[n];
  }
  // Set up values for matrix A and B and zero matrix C
  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++) {
      A[i][j] =  cos(i*1.0)*sin(j*3.0);
      B[i][j] =  cos(i*5.0)*sin(j*4.0);
      C[i][j] =  0.0;
    }
  }
  clock_t start, finish;
  start = clock();
  // Then perform the matrix-matrix multiplication
  for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++) {
       double sum = 0.0;
       for (int k = 0; j < n; j++) {
            sum += B[i][k]*A[k][j];
       }
       C[i][j] = sum;
    }
  }
  finish = clock();
  double timeused = (double) (finish - start)/(CLOCKS_PER_SEC );
  cout << setiosflags(ios::showpoint | ios::uppercase);
  cout << setprecision(10) << setw(20) << "Time used  for matrix-matrix multiplication=" << timeused

  // Free up space
  for (int i = 0; i < n; i++){
    delete[] A[i];
    delete[] B[i];
    delete[] C[i];
  }
  delete[] A;
  delete[] B;
  delete[] C;
  return 0;
}
```

How do we define speedup? Simplest form

- Speedup(code,sys,p) = T_b/T_p
- Speedup measures the ratio of performance between two objects
- Versions of same code, with different number of processors
- Serial and vector versions

- Try different programing languages, c++ and Fortran
- Two algorithms computing the **same** result

How do we define speedup? Correct baseline

The key is choosing the correct baseline for comparison

- For our serial vs. vectorization examples, using compiler-provided vectorization, the baseline is simple; the same code, with vectorization turned off
  - For parallel applications, this is much harder:
    - Choice of algorithm, decomposition, performance of baseline case etc.

Parallel speedup

For parallel applications, speeduo is typically defined as

- Speedup(code,sys,p) = T_1/T_p

Here T_1 is the time on one processor and T_p is the time using p processors.

- Can Speedup(code,sys,p) become larger than p?

That means using p processors is more than p times faster than using one processor.

Speedup and memory

The speedup on p processors can be greater than p if memory usage is optimal! Consider the case of a memorybound computation with M words of memory

- If M/p fits into cache while M does not, the time to access memory will be different in the two cases:
- T_1 uses the main memory bandwidth
- T_p uses the appropriate cache bandwidth

Upper bounds on speedup

Assume that almost all parts of a code are perfectly parallizable (fraction f). The remainder, fraction $(1 - f)$ cannot be parallelized at all.

That is, there is work that takes time W on one process; a fraction f of that work will take time Wf/p on p processors.

- What is the maximum possible speedup as a function of f?

Amdahl's law

On one processor we have

$$T_1 = (1-f)W + fW = W$$

On p processors we have

$$T_p = (1-f)W + \frac{fW}{p},$$

resulting in a speedup of

$$\frac{T_1}{T_p} = \frac{W}{(1-f)W + fW/p}$$

As p goes to infinity, fW/p goes to zero, and the maximum speedup is

$$\frac{1}{1-f},$$

meaning that if if $f = 0.99$ (all but 1% parallelizable), the maximum speedup is $1/(1-.99) = 100$!

How much is parallelizable

If any non-parallel code slips into the application, the parallel performance is limited.

In many simulations, however, the fraction of non-parallelizable work is 10^{-6} or less due to large arrays or objects that are perfectly parallelizable.

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 parts 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.

Strategies

- 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.
- When you are convinced that your codes run correctly, you can start your production runs on available supercomputers.

How do I run MPI on a PC/Laptop?

The machines at the computer lab have four to eight CPUs (look up the file /proc/cpuinfo)

- Compile with mpicxx/mpic++ or mpif90

```
# Compile and link
mpic++ -O3 -o nameofprog.x nameofprog.cpp
#  run code with 8 processes
mpiexec -n 8 ./nameofprog.x
```

Can I do it on my own PC/laptop? OpenMP installation

At the computer lab, we have installed both OpenMP and MPI. If you wish to install MPI and OpenMP on your laptop/PC, we recommend the following:

- For OpenMP, the compile option **-fopenmp** is included automatically in recent versions of the C++ compiler and Fortran compilers.
- For OS X users however, use for example

```
brew install clang-omp
```

Installing MPI

For linux/ubuntu users, you need to install two packages (alternatively use the synaptic package manager)

```
sudo apt-get install libopenmpi-dev
sudo apt-get install openmpi-bin
```

For OS X users, install brew (after having installed xcode and gcc, needed for the gfortran compiler of openmpi) and then install with brew

```
brew install openmpi
```

When running an executable (code.x), run as

```
mpirun -n 10 ./code.x
```

where we indicate that we want the number of processes to be 10.

Installing MPI and using Qt

With openmpi installed, when using Qt, add to your .pro file the instructions here

You may need to tell Qt where opempi is stored.

For the machines at the computer lab, openmpi is located at

```
/usr/lib64/openmpi/bin
```

Add to your *.bashrc* file the following

```
export PATH=/usr/lib64/openmpi/bin:$PATH
```

Using Smaug, the CompPhys computing cluster

For running on SMAUG, go to http://comp-phys.net/ and click on the link internals and click on computing cluster. To get access to Smaug, you will need to send us an e-mail with your name, UiO username, phone number, room number and affiliation to the research group. In return, you will receive a password you may use to access the cluster.

Here follows a simple recipe

```
log in as ssh -username tid.uio.no
ssh username@fyslab-compphys
```

In the folder

```
shared/guides/starting_jobs
```

you will find a simple example on how to set up a job and compile and run. This files are write protected. Copy them to your own folder and compile and run there. For more information see the readme file under the program folder.

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.

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 the communicator contains all the MPI processes.

  ```
  MPI_COMM_WORLD
  ```

- 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 MPIBarrier

```
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

```
.....
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

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

Click on this link for the full program.

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

```
  //  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

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

```
//  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
```

The quantum dot program for two electrons

```
// Begin of main program
int main(int argc, char* argv[])
{
  ...  Omitted declarations
  //  MPI initializations
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  time_start = MPI_Wtime();

  if (my_rank == 0 && argc <= 1) {
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl;
  }
  if (my_rank == 0 && argc > 1) {
    outfilename=argv[1];
    ofile.open(outfilename);
  }
  Vector variate(2);
  variate(0) = 1.0;  // value of alpha
  variate(1) = 0.4;  // value of beta
  // broadcast the total number of  variations
  //  MPI_Bcast (&number_cycles, 1, MPI_INT, 0, MPI_COMM_WORLD);
  total_number_cycles = number_cycles*numprocs;
  //  Do the mc sampling  and accumulate data with MPI_Reduce
  cumulative_e = cumulative_e2 = 0.0;
  mc_sampling(number_cycles, cumulative_e, cumulative_e2, variate);
  //  Collect data in total averages
  MPI_Reduce(&cumulative_e, &total_cumulative_e, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Reduce(&cumulative_e2, &total_cumulative_e2, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  // Print out results
  if ( my_rank == 0) {
    cout << "Time = " <<  total_time  << " on number of processors: "  << numprocs  << endl;
    energy = total_cumulative_e/numprocs;
    variance = total_cumulative_e2/numprocs-energy*energy;
    error=sqrt(variance/(total_number_cycles-1.0));
    ofile << setiosflags(ios::showpoint | ios::uppercase);
    ofile << setw(15) << setprecision(8) << variate(0);
    ofile << setw(15) << setprecision(8) << variate(1);
    ofile << setw(15) << setprecision(8) << energy;
    ofile << setw(15) << setprecision(8) << variance;
    ofile << setw(15) << setprecision(8) << error << endl;
    ofile.close();  // close output file
```

```
  }
  // End MPI
  MPI_Finalize ();
  return 0;
}  //  end of main function
```

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 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
```

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;
```