Exercises to support learning OpenMP[*]

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

Teaching Assistants:

Erin Carson (ecc2z@cs.berkeley.edu)

Nick Knight (knight@cs.berkeley.edu)

David Sheffield (dsheffie@cs.berkeley.edu)

[*] The name “OpenMP” is the property of the OpenMP Architecture Review Board.

Introduction

- This set of slides supports a collection of exercises to be used when learning OpenMP.
- Many of these are discussed in detail in our OpenMP tutorial. You can cheat and look up the answers, but challenge yourself and see if you can come up with the solutions on your own.
- A few (Exercise V, VI, and X) are more advanced. IF you are bored, skip directly to those problems. For exercise VI there are multiple solutions. Seeing how many different ways you can solve the problem is time well spent.

Acknowledgements

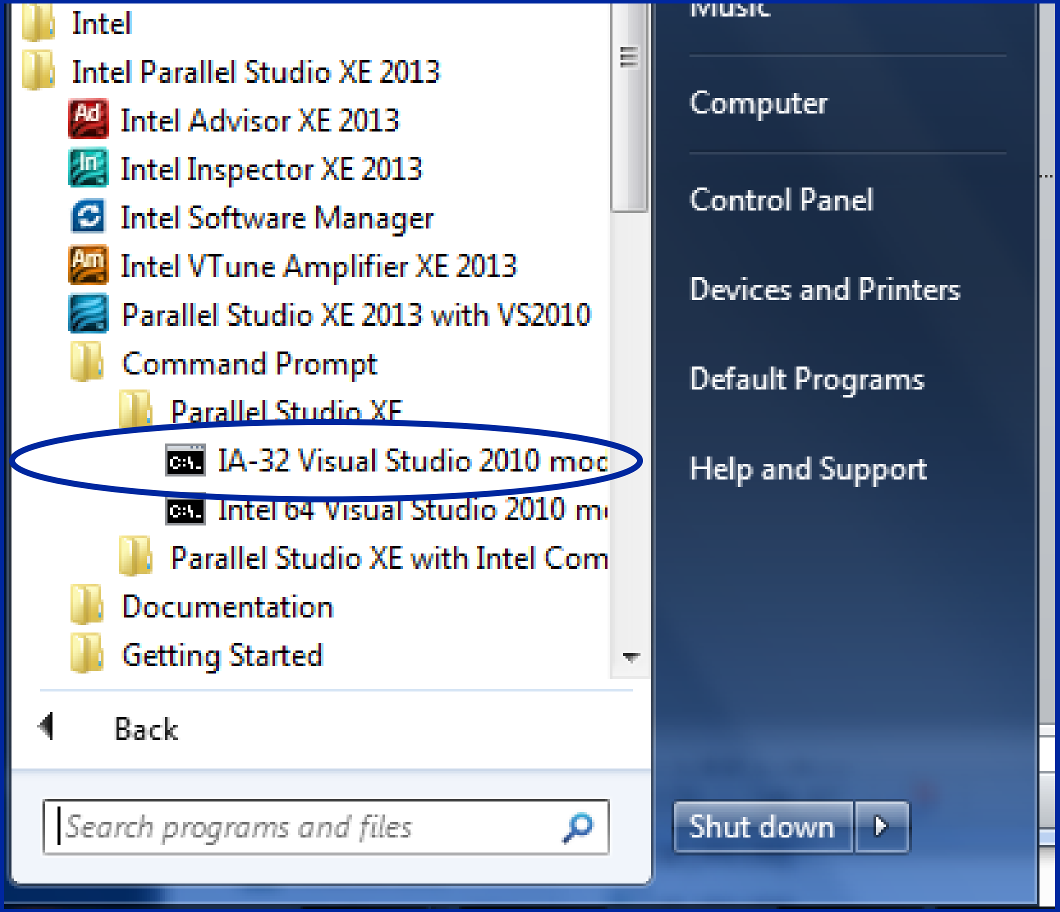
- Many people have worked on these exercises over the years.
- They are in the public domain and you can do whatever you like with them.
- Contributions from certain people deserve special notice:
  - Mark Bull (Mandelbrot set area),
  - Tim Mattson and Larry Meadows (Monte Carlo pi and random number generator)
  - Clay Breshears (recursive matrix multiplication).

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Compiler notes: Intel on Windows

- Launch SW dev environment
- cd to the directory that holds your source code
- Build software for program foo.c
  - ◆ icl /Qopenmp foo.c
- Set number of threads environment variable
  - ◆ set OMP_NUM_THREADS=4
- Run your program
  - ◆ foo.exe



To get rid of the "working directory name" on the prompt, type

prompt = %

Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language … and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

Compiler notes: Linux and OSX

- Linux and OS X with gcc:

```
> gcc -fopenmp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
```

for the Bash shell

- Linux and OS X with PGI:

```
> pgcc -mp foo.c
> export OMP_NUM_THREADS=4
> ./a.out
```

The gcc compiler provided with Xcode on OSX doesn't support the "threadprivate" construct and hence cannot be used for the "Monte Carlo Pi" exercise. "Monte Carlo pi" is one of the latter exercises, hence for most people this is not a problem.

Compiler notes: gcc on OSX

- To load a version of gcc with full OpenMP 3.1 support onto your mac running OSX, use the following steps:
  - Install mac ports from www.macports.org
  - Install gcc 4.8

    ```
    > sudo port install gcc48
    ```

  - Modify make.def in the OpenMP exercises directory to use the desired gcc compiler. On my system I need to change the CC definition line in make.def

    ```
    > CC = g++-mp-4.8
    ```

OpenMP constructs used in these exercises

- #pragma omp parallel
- #pragma omp for
- #pragma omp critical
- #pragma omp atomic
- #pragma omp barrier
- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)
- Tasks (remember … private data is made firstprivate by default)
  - pragma omp task
  - pragma omp taskwait
- #pragma threadprivate(variable_list)

Where variable_list is a comma separated list of variables

Print the value of the macro

_OPENMP

And its value will be

yyyymm

For the year and month of the spec the implementation used

Put this on a line right after you define the variables in question

Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{


     int ID = 0;

     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include <omp.h>
int main()
{
 #pragma omp parallel
 {

    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

Switches for compiling and linking

g++ -fopenmp   Linus, OSX

pgcc -mp pgi

icl /Qopenmp intel (windows)

icpc –openmp intel (linux)

Solution

Exercise 1: Solution

A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
int  main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

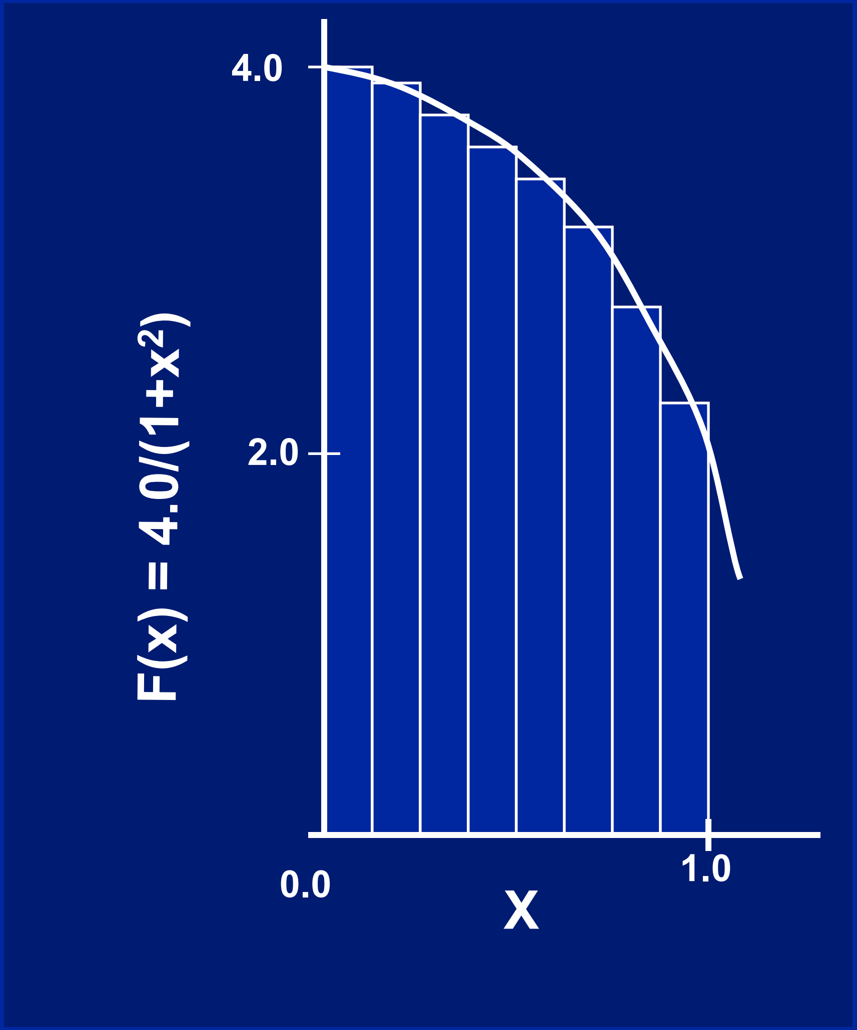
```
hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)
```

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercises 2 to 4:

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Exercises 2 to 4: Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{           int i;   double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

See OMP_exercises/pi.c

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

The SPMD pattern

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region “near the top of the code”.
  - Pick up thread ID and num_threads.
  - Use them to split up loops and select different blocks of data to work on.

Solution

Exercise 2: A simple SPMD pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{             int i, nthreads;  double pi, sum[NUM_THREADS];
              step = 1.0/(double) num_steps;
              omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
             int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                          x = (i+0.5)*step;
                          sum[id] += 4.0/(1.0+x*x);
              }
    }
              for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
➡ III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 3

- In exercise 2, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence “sloshing independent data” back and forth between threads.
- Modify your “pi program” from exercise 2 to avoid false sharing due to the sum array.

False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads.
  - This is called “false sharing”.
- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Solution

Exercise 3: SPMD Pi without false sharing

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{            double  pi;          step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
            int i, id,nthrds;    double x, sum;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
             id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
             for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
             }
          #pragma omp critical
                  pi += sum * step;
}
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes “out of scope” beyond the parallel region … so you must sum it in here.   Must protect summation into pi in a critical region so updates don’t conflict

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 4: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Solution

Exercise 4: solution

```
#include <omp.h>
static long num_steps = 100000;         double step;
void main ()
{    int i;              double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
          double x;
        #pragma omp for reduction(+:sum)
              for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
              }
      }
            pi = step * sum;
}
```

Exercise 4: solution

Using data environment clauses so parallelization only requires changes to the pragma

```
#include <omp.h>
static long num_steps = 100000;         double step;

void main ()
{          int i;     double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
           for (i=0;i< num_steps; i++){
                    x = (i+0.5)*step;
                    sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 2 simple lines of text!

Exercise 5: Optimizing loops

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?

Solution

Matrix multiplication

```
#pragma omp parallel for private(tmp, i, j, k)
   for (i=0; i<Ndim; i++){
         for (j=0; j<Mdim; j++){
                  tmp = 0.0;
                  for(k=0;k<Pdim;k++){
                           /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
                           tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
                  }
                  *(C+(i*Ndim+j)) = tmp;
         }
   }
```

- On a dual core laptop
  - 13.2 seconds 153 Mflops one thread
  - 7.5 seconds 270 Mflops two threads

Results on an Intel dual core 1.83 GHz CPU,   Intel IA-32 compiler 10.1 build 2

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 6: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn’t do it right.
- Find and fix the errors (hint … the problem is with the data environment).

Exercise 6 (cont.)

- Once you have a working version, try to optimize the program?
  - Try different schedules on the parallel loop.
  - Try different mechanisms to support mutual exclusion … do the efficiencies change?

Solution

The Mandelbrot Area program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
void testpoint(void);
struct d_complex{
   double r;     double i;
};
struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) private(c,eps)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint();
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(void){
struct d_complex z;
       int iter;
       double temp;

       z=c;
       for (iter=0; iter<MXITR; iter++){
         temp = (z.r*z.r)-(z.i*z.i)+c.r;
         z.i = z.r*z.i*2+c.i;
         z.r = temp;
         if ((z.r*z.r+z.i*z.i)>4.0) {
           numoutside++;
           break;
         }
       }
}
```

When I run this program, I get a different incorrect answer each time I run it … there is a race condition!!!!

Debugging parallel programs

- Find tools that work with your environment and learn to use them.   A good parallel debugger can make a huge difference.
- But parallel debuggers are not portable and you will assuredly need to debug “by hand” at some point.
- There are tricks to help you. The most important is to use the default(none) pragma

```
#pragma omp parallel for default(none) private(c, eps)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint();
     }
   }
}
```

Using default(none) generates a compiler error that j is unspecified.

Area of a Mandelbrot set

- Solution is in the file mandel_par.c
- Errors:
  - Eps is private but uninitialized.   Two solutions
    - It’s read-only so you can make it shared.
    - Make it firstprivate
  - The loop index variable j is shared by default. Make it private.
  - The variable c has global scope so “testpoint” may pick up the global value rather than the private value in the loop. Solution … pass C as an arg to testpoint
  - Updates to “numoutside” are a race. Protect with an atomic.

The Mandelbrot Area program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
struct d_complex{
   double r;     double i;
};
void testpoint(struct d_complex);
struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
    firstpriivate(eps)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint(c);
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(struct  d_complex c){
struct d_complex z;
       int iter;
       double temp;

       z=c;
       for (iter=0; iter<MXITR; iter++){
         temp = (z.r*z.r)-(z.i*z.i)+c.r;
         z.i = z.r*z.i*2+c.i;
         z.r = temp;
         if ((z.r*z.r+z.i*z.i)>4.0) {
         #pragma omp atomic
           numoutside++;
           break;
         }
       }
}
```

Other errors found using a debugger or by inspection:

- **eps was not initialized**
- **Protect updates of numoutside**
- **Which value of c did testpoint() see? Global or private?**

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

list traversal

- When we first created OpenMP, we focused on common use cases in HPC … Fortran arrays processed over “regular” loops.
- Recursion and “pointer chasing” were so far removed from our Fortan focus that we didn’t even consider more general structures.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

```
 p=head;
 while (p) {
    process(p);
    p = p->next;
}
```

Exercise 7: linked lists the hard way

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using constructs defined in worksharing constructs … i.e. don’t use tasks).
- Once you have a correct program, optimize it.

Solution

Linked lists without tasks

- See the file Linked_omp25.c

```
while (p != NULL) {
     p = p->next;
      count++;
}
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
     #pragma omp for schedule(static,1)
     for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU,   Intel IA-32 compiler 10.1 build 2

Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

```
 std::vector<node *> nodelist;
 for (p = head; p != NULL; p = p->next)
   nodelist.push_back(p);

int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
   for (int i = 0; i < j; ++i)
        processwork(nodelist[i]);
```

Copy pointer to each node into an array

Count number of items in the linked list

Process nodes in parallel with a for loop

	C++, default sched.	C++, (static,1)	C, (static,1)
One Thread	37 seconds	49 seconds	45 seconds
Two Threads	47 seconds	32 seconds	28 seconds

Results on an Intel dual core 1.83 GHz CPU,   Intel IA-32 compiler 10.1 build 2

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 8: tasks in OpenMP

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.
- Compare your solution's complexity to an approach without tasks.

Linked lists with tasks (OpenMP 3)

- See the file Linked_omp3_tasks.c

```
#pragma omp parallel
{
  #pragma omp single
  {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
             processwork(p);
         p = p->next;
    }
  }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined

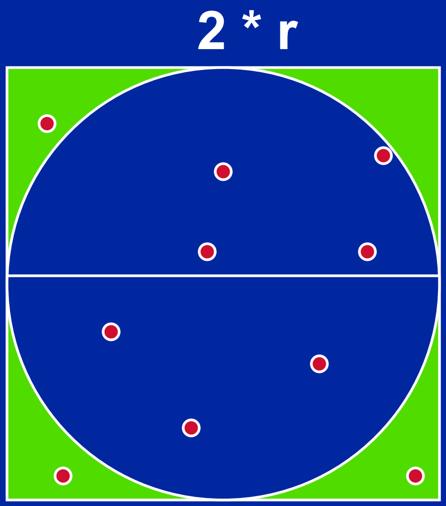
OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 9: Monte Carlo Calculations

Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Exercise 9

- We provide three files for this exercise
  - pi_mc.c: the monte carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be threadsafe.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

Solution

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
   long i;      long Ncirc = 0;       double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(0,-r, r);  // The circle and square are centered at the origin
   #pragma omp parallel for private (x, y) reduction (+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
      x = random();        y = random();
      if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Computers and random numbers

- We use "dice" to make random numbers:
  - Given previous values, you cannot predict the next value.
  - There are no patterns in the series … and it goes on forever.
- Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  - By design, computers are not random and cannot produce random numbers.
- However, with some very clever programming, we can make "pseudo random" numbers that are as random as you need them to be … but only if you are very careful.
- Why do I care? Random numbers drive statistical methods used in countless applications:
  - Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

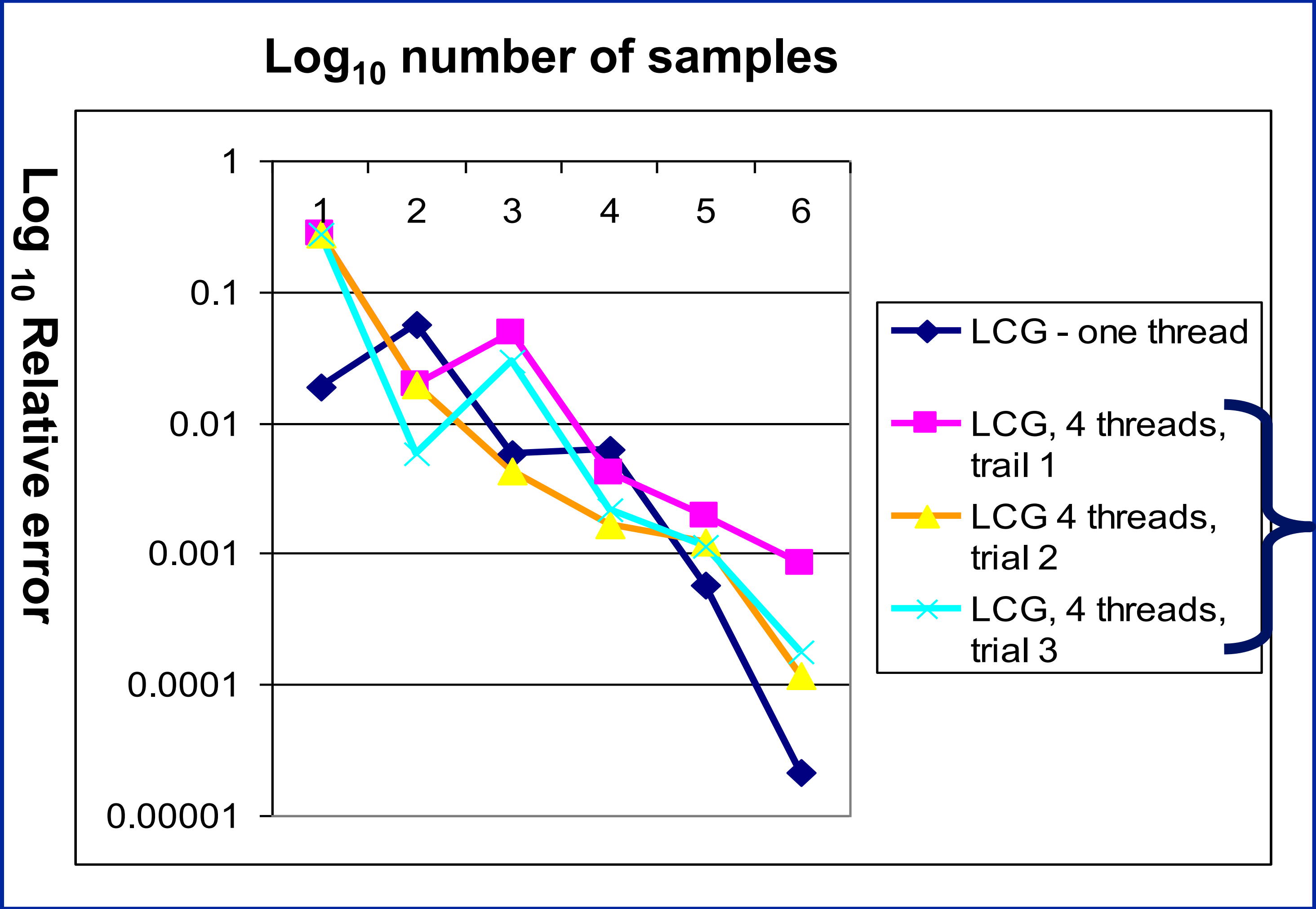
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

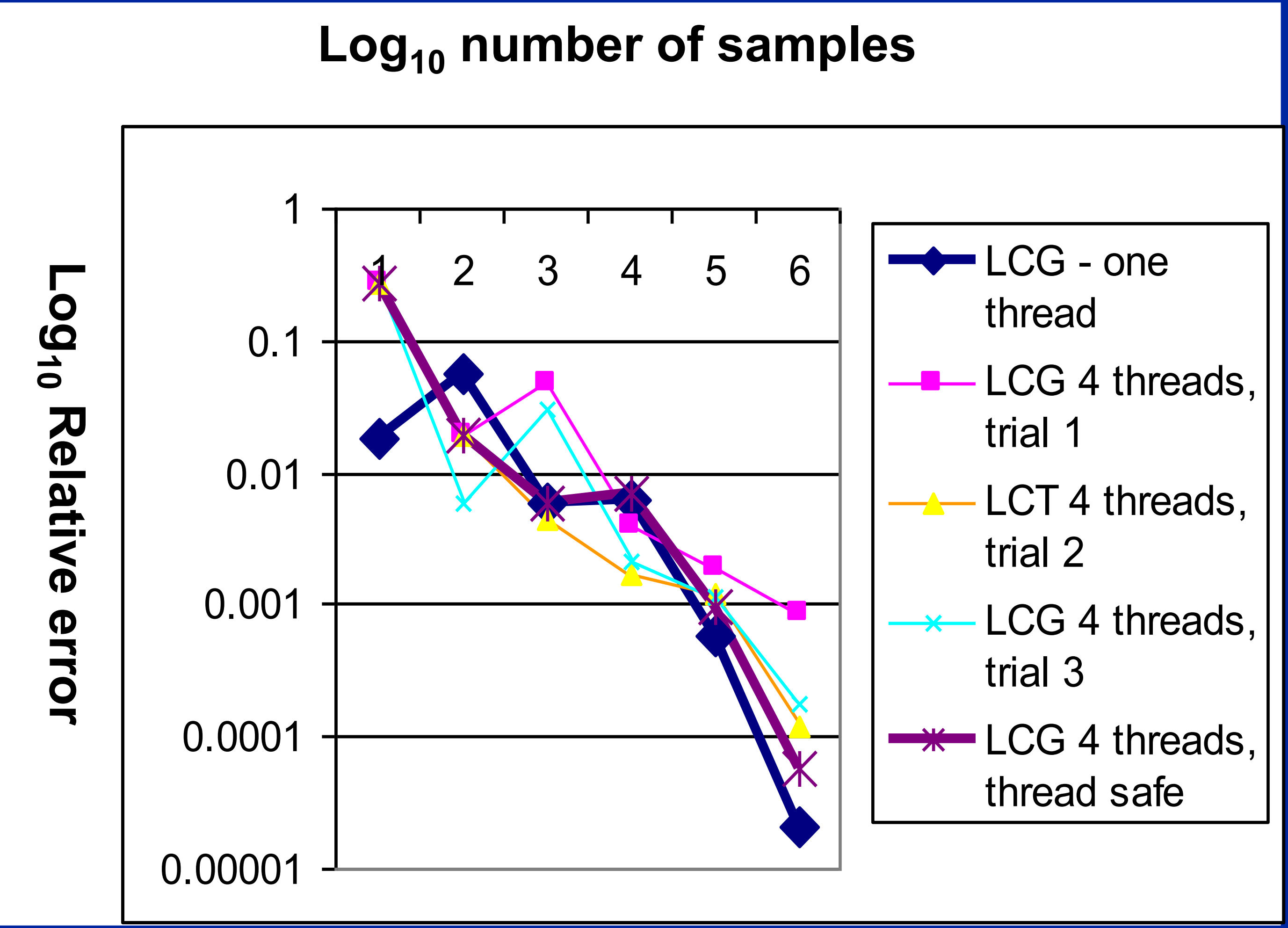
LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
   random_last = random_next;

  return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread safe random number generators



Thread safe version gives the same answer each time you run the program.

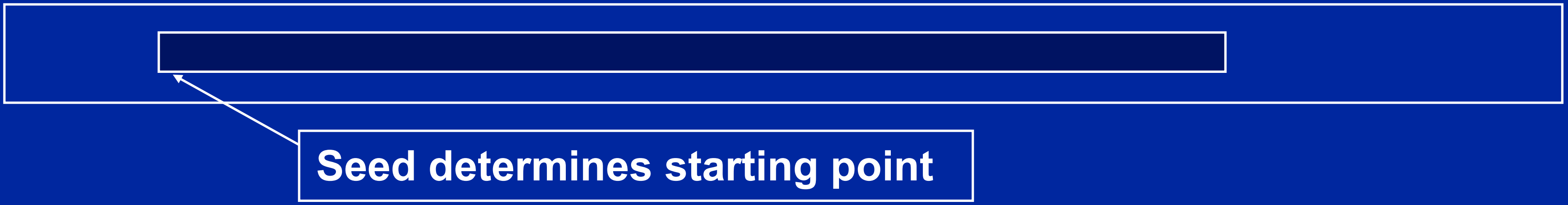
But for large number of samples, its quality is lower than the one thread result!

Why?

Pseudo Random Sequences

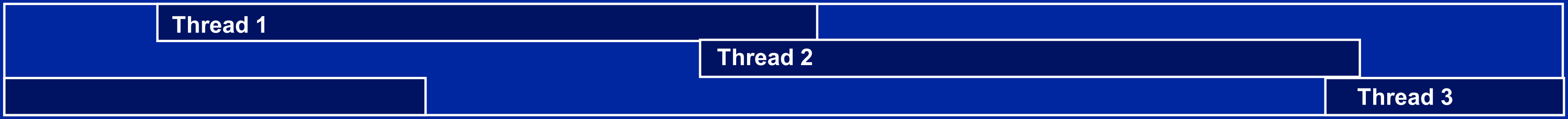
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range



Seed determines starting point

- Grab arbitrary seeds and you may generate overlapping sequences
  - E.g. three sequences … last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Replicate and Pray
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values “round robin” as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.
- Other than “replicate and pray”, these are difficult to implement. Be smart … buy a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Nice for debugging, but not really needed scientifically.

Intel’s Math kernel Library supports all of these methods.

MKL Random number generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                    BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Select type of RNG and set seed

Initialize a stream or pseudo random numbers

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

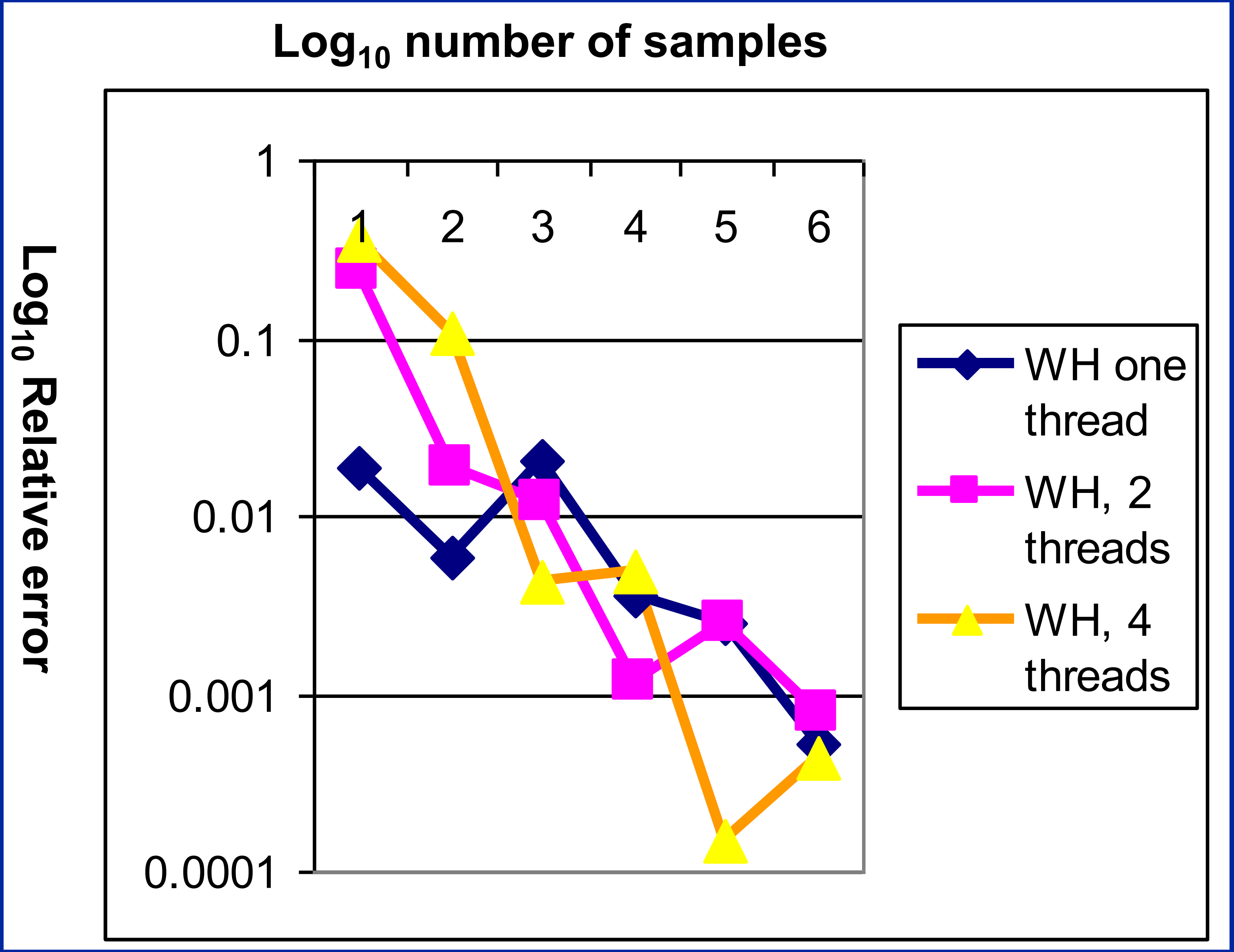
Delete the stream when you are done

Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                              …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the ith number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
     iseed = PMOD/MULTIPLIER;     // just pick a seed
     pseed[0] = iseed;
     mult_n = MULTIPLIER;
     for (i = 1; i < nthreads; ++i)
    {
         iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
         pseed[i] = iseed;
         mult_n = (mult_n * MULTIPLIER) % PMOD;
    }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Exercise 10: producer consumer

- Parallelize the “prod_cons.c” program.
- This is a well known pattern called the producer consumer pattern
  - One thread produces values that another thread consumes.
  - Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

Exercise 10: prod_cons.c

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *)malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
}
```

Pair wise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Solution

Exercise 10: producer consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));

   #pragma omp parallel sections
   {
     #pragma omp section
      {
         fill_rand(N, A);
         #pragma omp flush
         flag = 1;
         #pragma omp flush (flag)
      }
      #pragma omp section
      {
         #pragma omp flush (flag)
         while (flag != 1){
             #pragma omp flush (flag)
         }
         #pragma omp flush
         sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the “produced” value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of A

Flush needed on both “reader” and “writer” sides of the communication

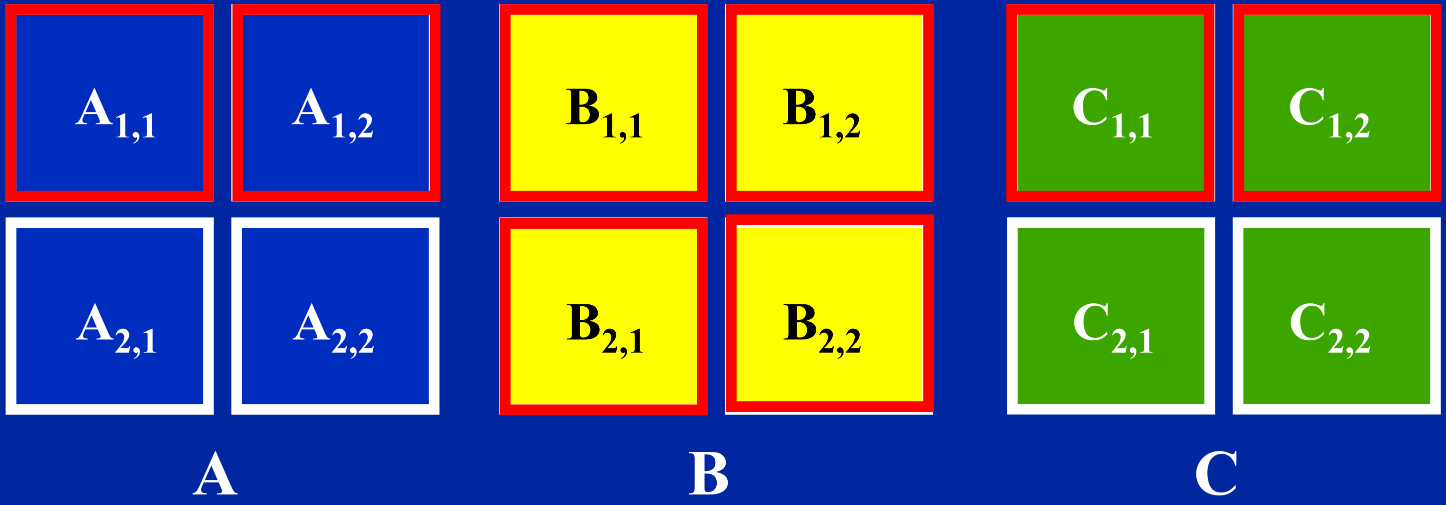
Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

OpenMP Exercises

Topic	Exercise	concepts
I. OMP Intro	Install sw, hello_world	Parallel regions
II. Creating threads	Pi_spmd_simple	Parallel, default data environment, runtime library calls
III. Synchronization	Pi_spmd_final	False sharing, critical, atomic
IV. Parallel loops	Pi_loop, Matmul	For, schedule, reduction,
V. Data Environment	Mandelbrot set area	Data environment details, software optimization
VI. Practice with core OpenMP constructs	Traverse linked lists … the old fashioned way	Working with more complex data structures with parallel regions and loops
VII. OpenMP tasks	Traversing linked lists	Explicit tasks in OpenMP
VIII. ThreadPrivate	Monte Carlo pi	Thread safe libraries
IX: Pairwise synchronization	Producer Consumer	Understanding the OpenMP memory model and using flush
X: Working with tasks	Recursive matrix multiplication	Explicit tasks in OpenMP

Recursive matrix multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions



$A_{1,1}$ $A_{1,2}$ $A_{2,1}$ $A_{2,2}$

$B_{1,1}$ $B_{1,2}$ $B_{2,1}$ $B_{2,2}$

$C_{1,1}$ $C_{1,2}$ $C_{2,1}$ $C_{2,2}$

A B C

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

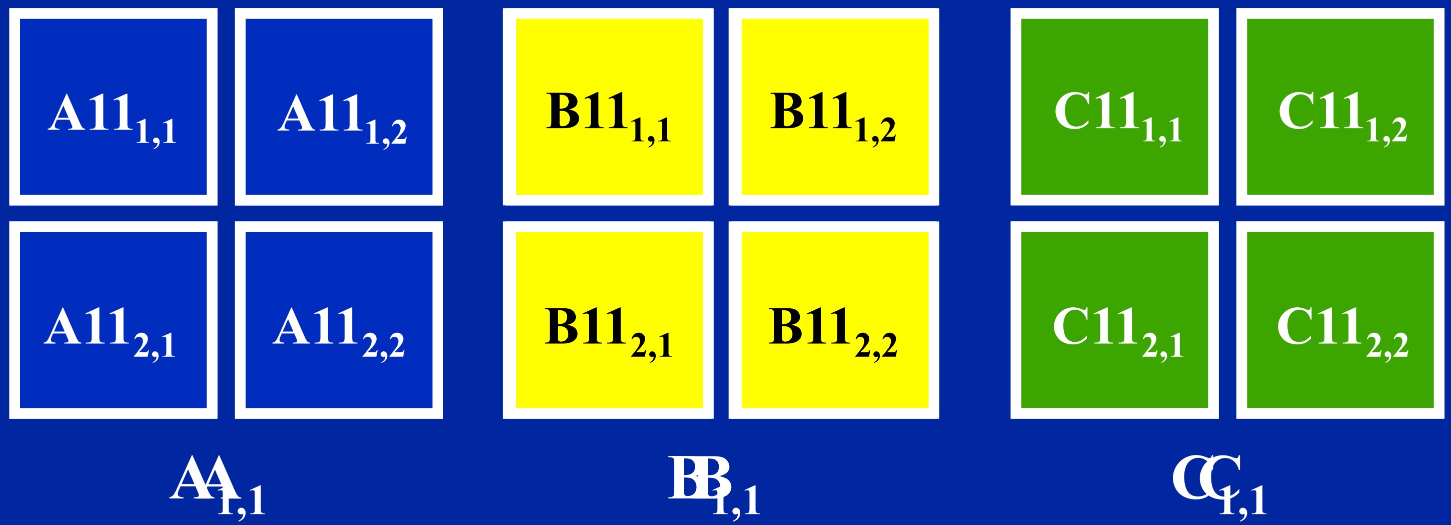
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

How to multiply submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply



$A11_{1,1}$ $A11_{1,2}$ $B11_{1,1}$ $B11_{1,2}$ $C11_{1,1}$ $C11_{1,2}$

$A11_{2,1}$ $A11_{2,2}$ $B11_{2,1}$ $B11_{2,2}$ $C11_{2,1}$ $C11_{2,2}$

A $A_{1,1}$ B $B_{1,1}$ C $C_{1,1}$

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

$$C11_{1,1} = A11_{1,1} \cdot B11_{1,1} + A11_{1,2} \cdot B11_{2,1} + A12_{1,1} \cdot B21_{1,1} + A12_{1,2} \cdot B21_{2,1}$$

Recursively multiply submatrices

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \qquad C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \qquad C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

- Need range of indices to define each submatrix to be used

```
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

// C11 += A11*B11
     matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);
// C11 += A12*B21
     matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);
   . . .
}
```

- Also need stopping criteria for recursion

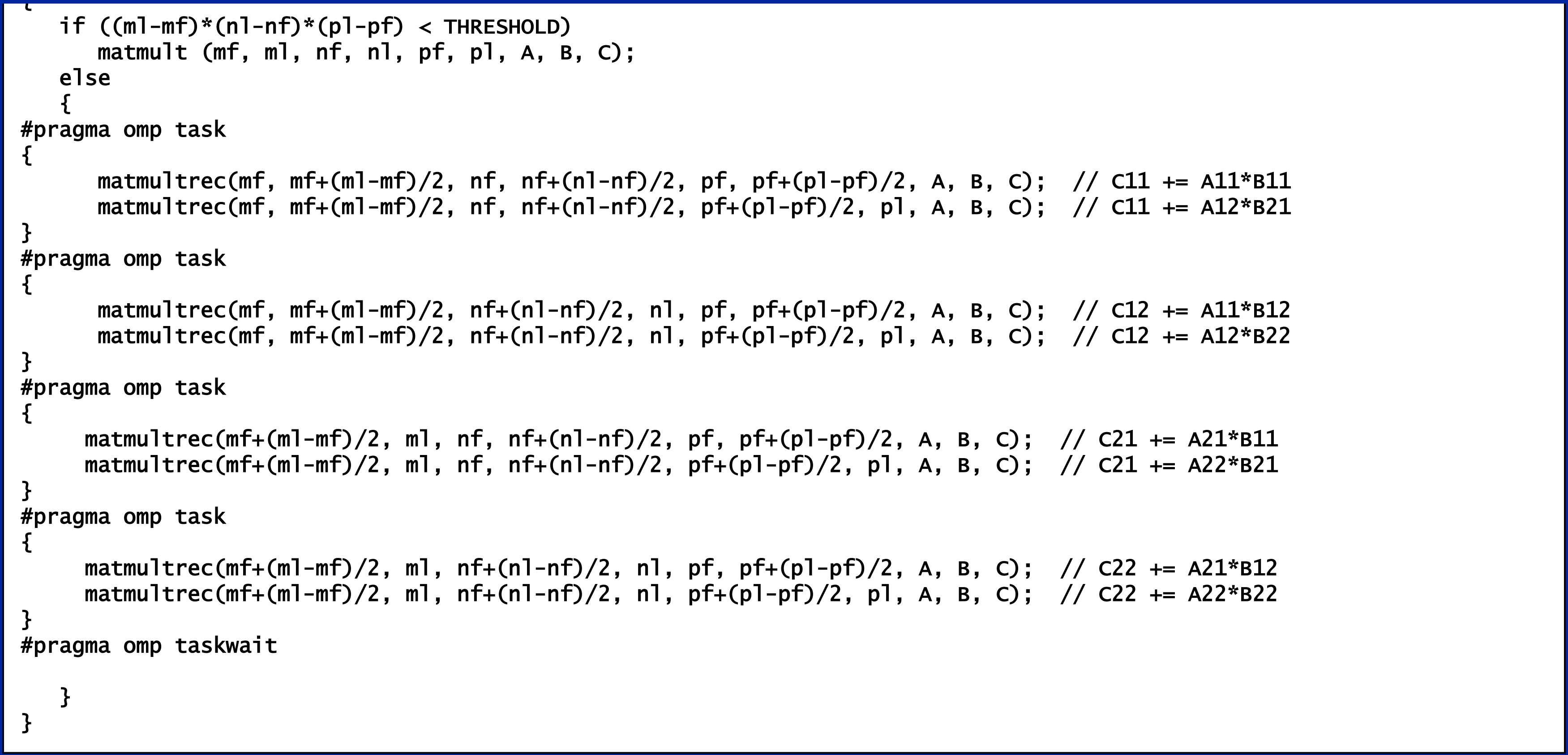
Exercise 11: Recursive matrix multiplication

- Consider the program matmul_recur.c. This program implements a recursive algorithm for multiply two square matrices.
  - Parallelize this program using OpenMP tasks.
  - Optimize the program. How does performance with the optimized version of the program compare to the loop-based program from exercise 5

Solution

Recursive Solution

- Could be executed in parallel as 4 tasks
  - ◆ Each task executes the two calls for the same output submatrix of C
- However, the same number of multiplication operations needed



```
{
   if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
      matmult (mf, ml, nf, nl, pf, pl, A, B, C);
   else
   {
#pragma omp task
{
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
}
#pragma omp task
{
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
}
#pragma omp task
{
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
}
#pragma omp task
{
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
}
#pragma omp taskwait

   }
}
```