OpenMP™

Challenge problems … so no has any excuse to get bored

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

J. Mark Bull

EPCC, The University of Edinburgh

markb@epcc.ed.ac.uk

Challenge Problems

- Long term retention of acquired skills is best supported by “random practice”.
  - i.e. a set of exercises where you must draw on multiple facets of the skills you are learning.
- To support “Random Practice” we have assembled a set of “challenge problems”
  1. Parallel Molecular dynamics
  2. Monte Carlo “pi” program and parallel random number generators
  3. Optimizing matrix multiplication
  4. Traversing linked lists in different ways
  5. Recursive matrix multiplication algorithms
  6. Pairwise synchronization (producer-consumer)

Outline

- OpenMP Content to support the Challenge problems:
  - ➡ Threadprivate Data
  - Flush and Atomic constructs
- Challenge Problems
- Solutions

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

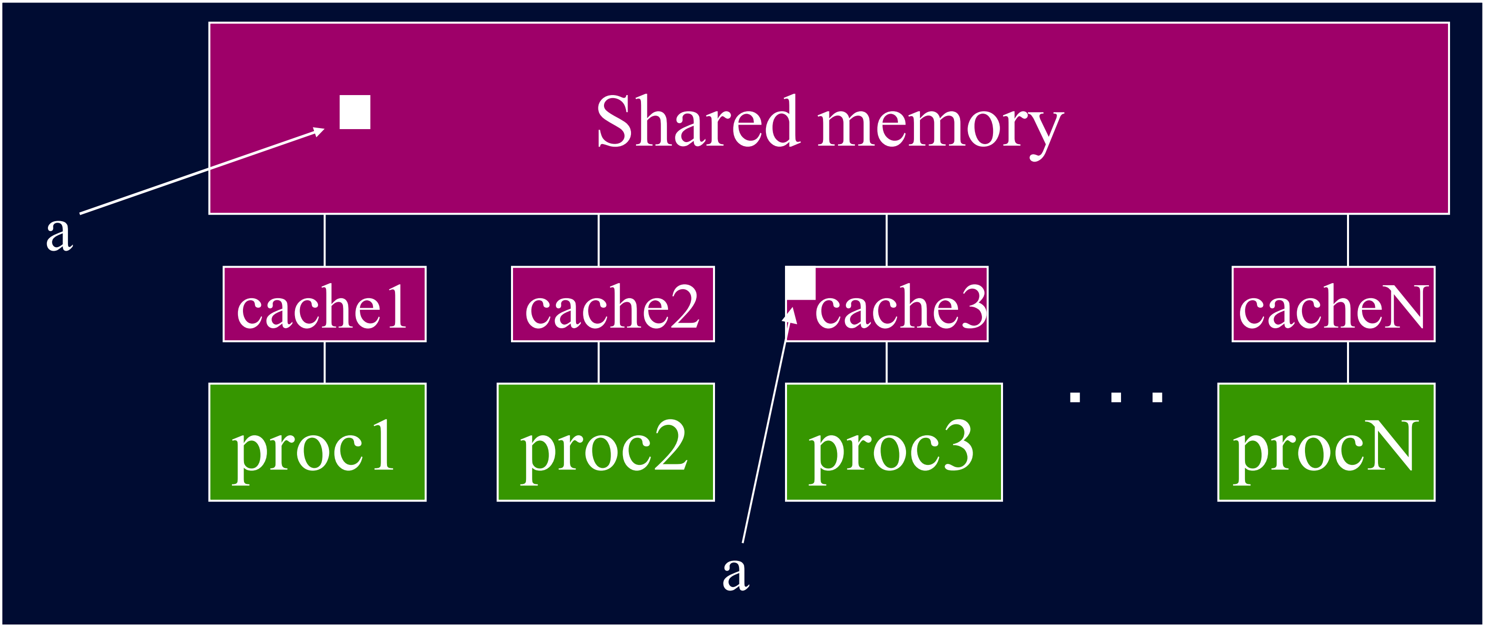
```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

Outline

- OpenMP Content to support the Challenge problems:
    - Threadprivate Data
    - ➡ Flush and Atomic constructs
- Challenge Problems
- Solutions

OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- Multiple copies of data may be present in various levels of cache, or in registers.

OpenMP and Relaxed Consistency

- OpenMP supports a relaxed-consistency shared memory model.
  - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
  - These temporary views are made consistent only at certain points in the program.
  - The operation which enforces consistency is called the flush operation

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred
  - A flush operation is analogous to a fence in other shared memory API’s

Synchronization: flush example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

#pragma omp flush(A)

   // flush to memory to make sure other
   //  threads can pick up the right value
```

Note: OpenMP’s flush is analogous to a fence in other shared memory API’s.

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- Compiler generally cannot move instructions:
  - past a barrier
  - past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if “flush(list)” is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s values are made consistent with main memory.

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

Synchronization: Atomic

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel

{
         double tmp, B;

        B =  DOIT();

        tmp = big_ugly(B);

 #pragma omp atomic
           X +=  tmp;

}
```

Atomic only protects the read/update of X

The OpenMP 3.1 atomics (1 of 2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

```
# pragma omp atomic [read | write | update | capture]
```

- Atomic can protect loads

```
# pragma omp atomic read
        v = x;
```

- Atomic can protect stores

```
# pragma omp atomic write
        x = expr;
```

- Atomic can protect updates to a storage location (this is the default behavior … i.e. when you don't provide a clause)

```
# pragma omp atomic update
        x++;  or ++x;  or x--;  or –x;  or
        x binop= expr; or x = x binop expr;
```

This is the original OpenMP atomic

The OpenMP 3.1 atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:

```
# pragma omp atomic capture
            statement or structured block
```

- Where the statement is one of the following forms:

```
v = x++;     v = ++x;     v = x--;     v =  –x;     v = x binop expr;
```

- Where the structured block is one of the following forms:

```
{v = x;  x binop = expr;}           {x  binop = expr;     v = x;}
{v=x;    x=x binop expr;}           {X = x binop expr;   v = x;}
{v = x;   x++;}                     {v=x;     ++x:}
{++x;     v=x:}                     {x++;      v = x;}
{v = x;    x--;}                    {v= x;     --x;}
{--x;       v = x;}                 {x--;       v = x;}
```

The capture semantics in atomic were added to map onto common hardware supported atomic ops and to support modern lock free algorithms.

Outline

- OpenMP Content to support the Challenge problems:
  - Threadprivate Data
  - Atomic construct
- ➡ Challenge Problems
- Solutions

Challenge 1: Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine `forces` (in forces.c)
- Parallelise this routine using a parallel for construct and atomics. Think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables.
- Experiment with different schedules kinds.

Challenge 1: MD (cont.)

- Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  - code other than the forces loop must be executed by a single thread (or workshared).
  - how does the data sharing change?
- The atomics are a bottleneck on most systems.
  - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number.
  - Which thread(s) should do the final accumulation into f?

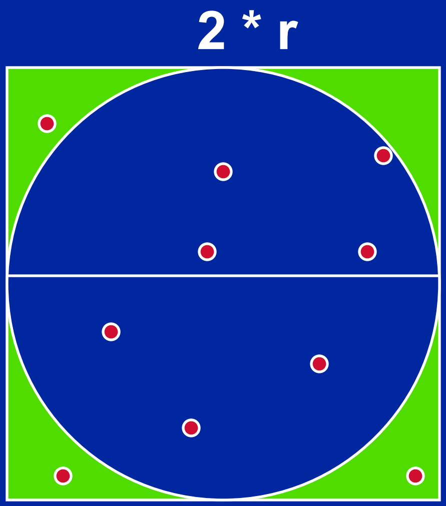
Challenge 1 MD: (cont.)

- Another option is to use locks
  - Declare an array of locks
  - Associate each lock with some subset of the particles
  - Any thread which is updating the force on a particle must hold the corresponding lock
  - Try to avoid unnecessary acquires/releases
  - What is the best number of particles per lock?

Challenge 2: 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.

Challenge 2: Monte Carlo pi (cont)

- 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).

Challenge 3: Matrix Multiplication

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?

Challenge 4: traversing linked lists

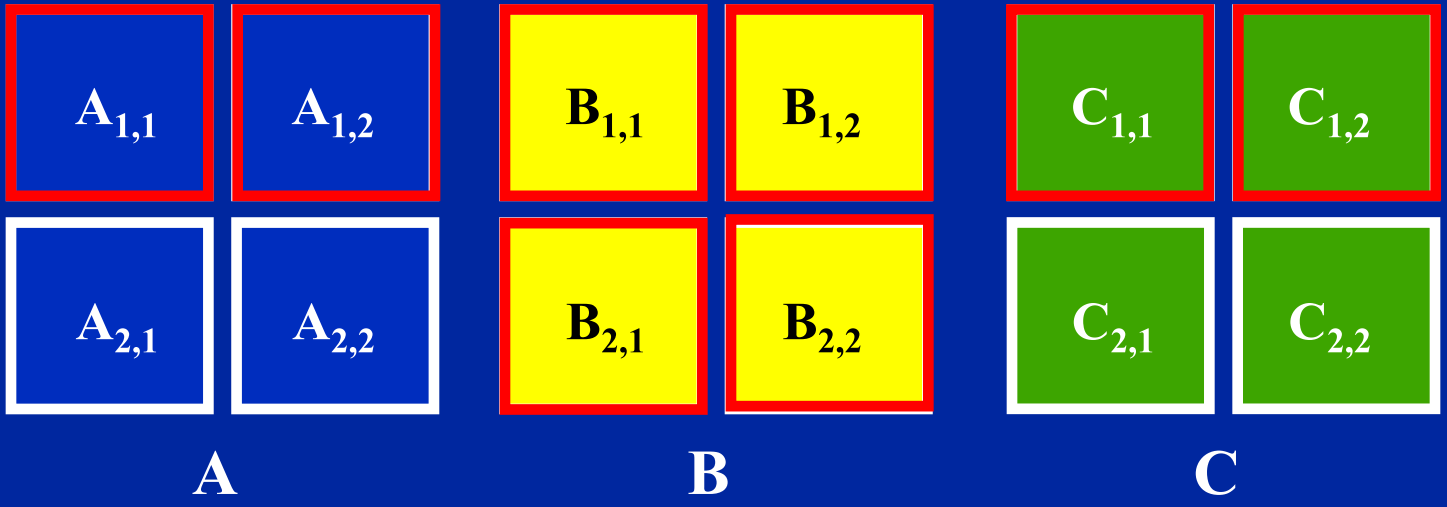
- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).

Challenge 5: Recursive matrix multiplication

- The following three slides explain how to use a recursive algorithm to multiply a pair of matrices.
- Source code implementing this algorithm is provided in the file matmul_recur.c.
- Parallelize this program using OpenMP tasks.

Challenge 5: Recursive matrix multiplication

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



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

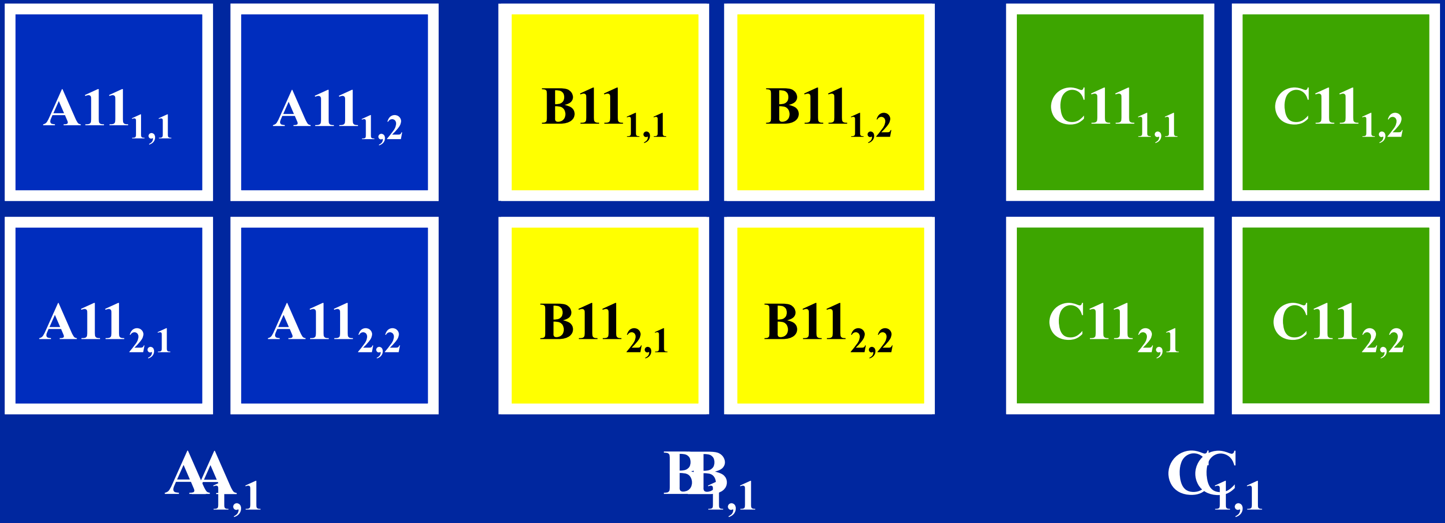
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

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

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

Challenge 5: Recursive matrix multiplication
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



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

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

Challenge 5: Recursive matrix multiplication 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

Challenge 6: Producer-consumer

- Consider the program prod_cons
  - Two functions: one fills an array with random numbers and the second sums the array.
- Parallelize this program so it runs on two threads.
  - First thread: the producer … fills the array
  - Second Thread: he consumer … sums the array
- Implemente pairwise synchronization in OpenMP so the program produces the correct results without any data races

Outline

- OpenMP Content to support the Challenge problems:
  - Threadprivate Data
  - Atomic construct
- Challenge Problems
- ➡ Solutions

Challenge Problem Solutions

- → Challenge 1: molecular dynamics
- Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- Challenge 6: Producer-consumer

Challenge 1: solution

Compiler will warn you if you have missed some variables

```
#pragma omp parallel for default (none) \
    shared(x,f,npart,rcoff,side) \
    reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
     ………
```

Loop is not well load balanced: best schedule has to be found by experiment.

Challenge 1: solution (cont.)

```
……..
#pragma omp atomic
     f[j]    -= forcex;
#pragma omp atomic
     f[j+1]  -= forcey;
#pragma omp atomic
     f[j+2]  -= forcez;
    }
   }
#pragma omp atomic
   f[i]     += fxi;
#pragma omp atomic
   f[i+1]   += fyi;
#pragma omp atomic
   f[i+2]   += fzi;
  }
 }
```

All updates to f must be atomic

Challenge 1: with orphaning

```
#pragma omp single
{
   vir    = 0.0;
   epot   = 0.0;
}
#pragma omp for reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
……….
```

Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

All variables which used to be shared here are now implicitly determined

Challenge 1: with array reduction

```
      ftemp[myid][j]    -= forcex;
      ftemp[myid][j+1]  -= forcey;
      ftemp[myid][j+2]  -= forcez;
     }
    }
   ftemp[myid][i]       += fxi;
   ftemp[myid][i+1]     += fyi;
   ftemp[myid][i+2]     += fzi;
  }
```

Replace atomics with accumulation into array with extra dimension

Challenge 1: with array reduction

```
….
#pragma omp for
    for(int i=0;i<(npart*3);i++){
        for(int id=0;id<nthreads;id++){
            f[i] += ftemp[id][i];
            ftemp[id][i] = 0.0;
        }
    }
```

Reduction can be done in parallel

Zero ftemp for next time round

Challenge Problem Solutions

- Challenge 1: molecular dynamics
- ➡ Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- Challenge 6: Producer-consumer

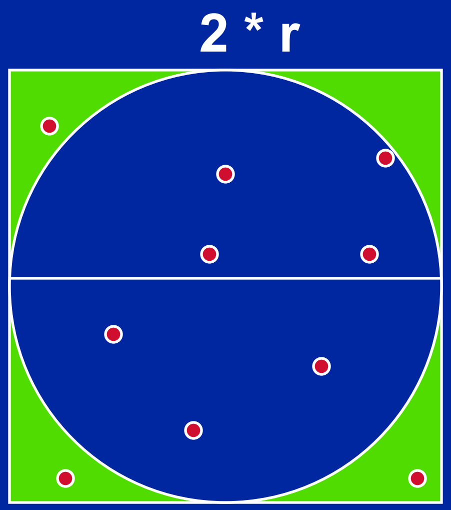
Computers and random numbers

- 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).

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.

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);
}
```

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;
```

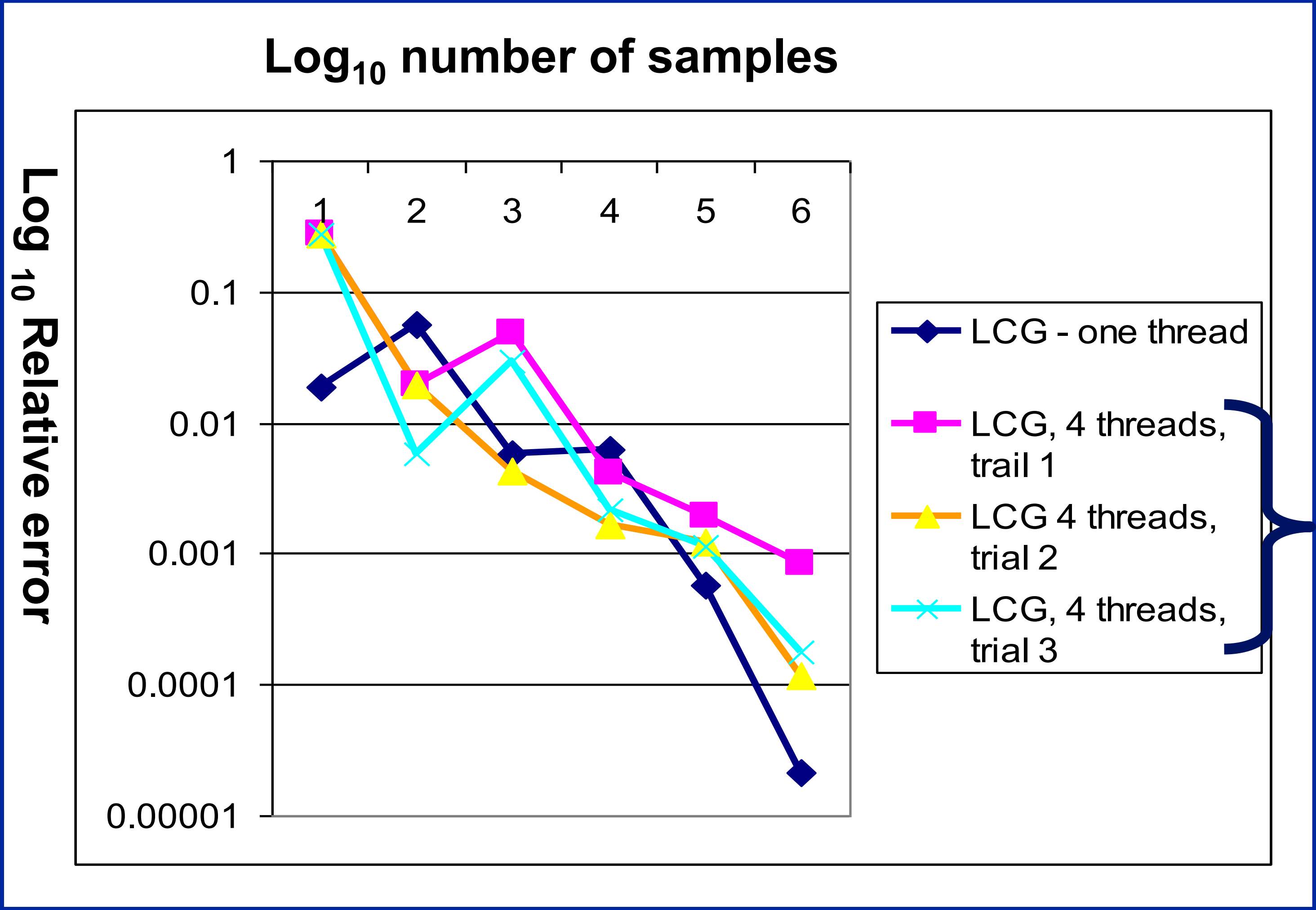
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

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.

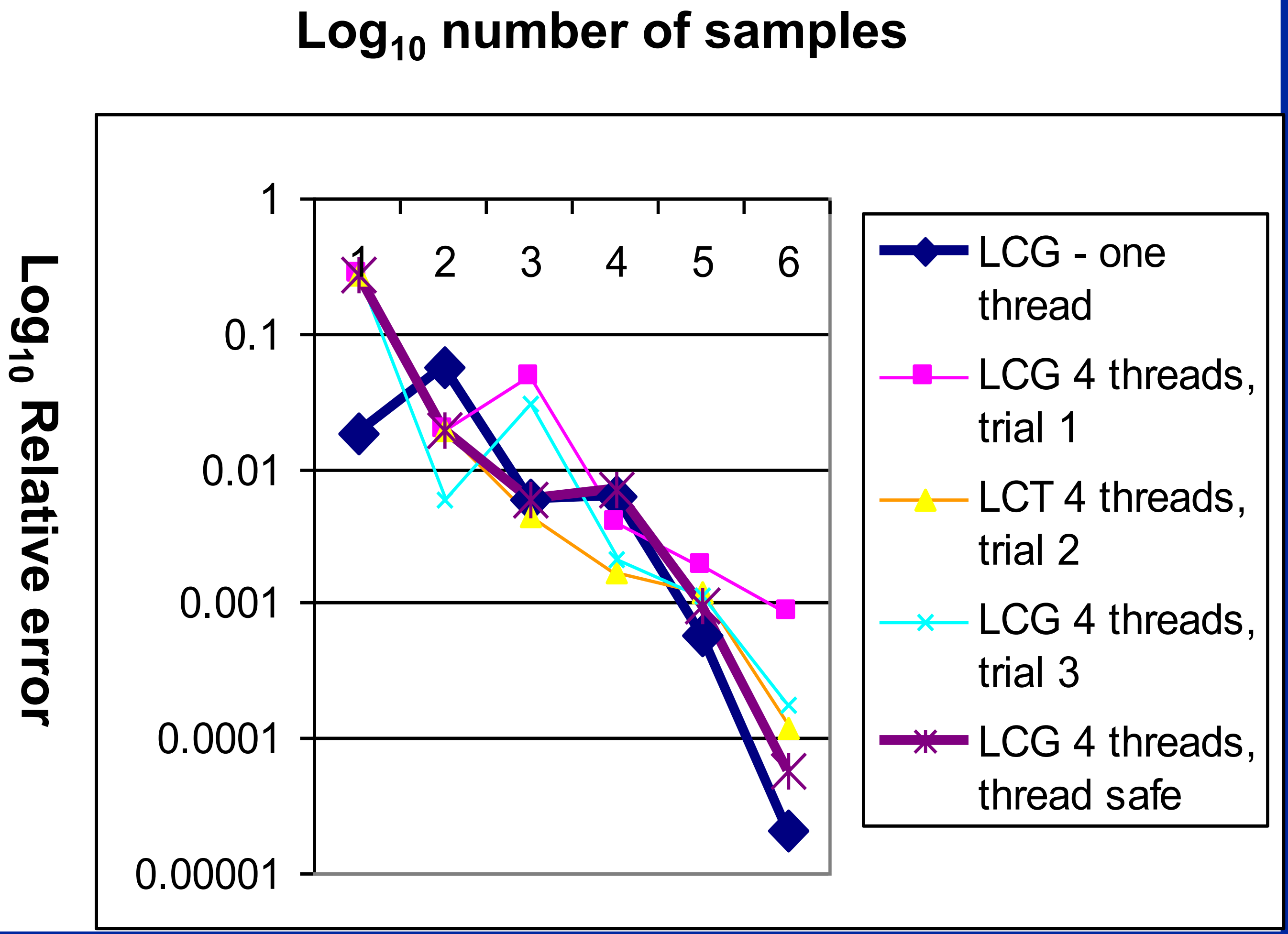
LCG code: threadsafe version

```
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.

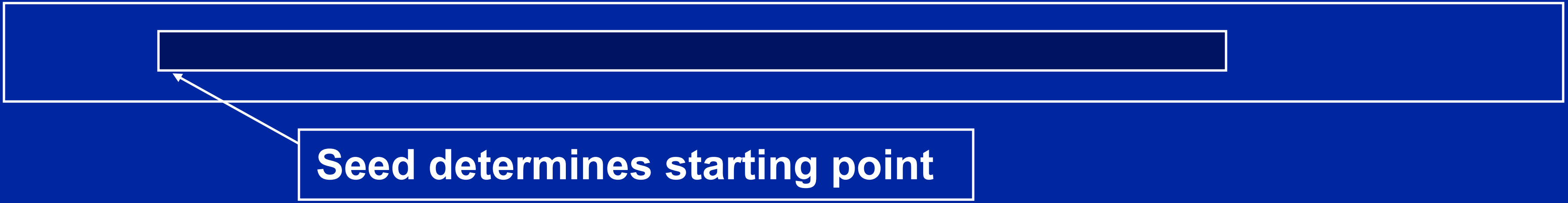
But for large number of samples, its quality is lower than the one thread result!

Why?

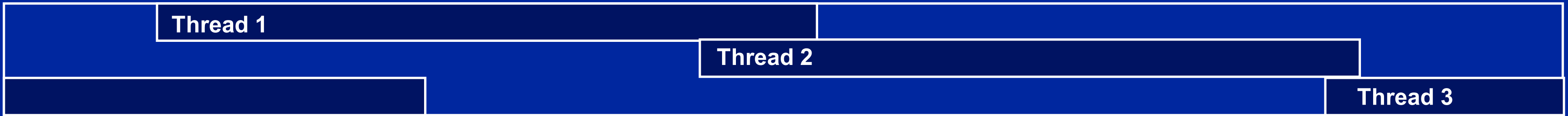
Pseudo Random Sequences

- 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



- 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.

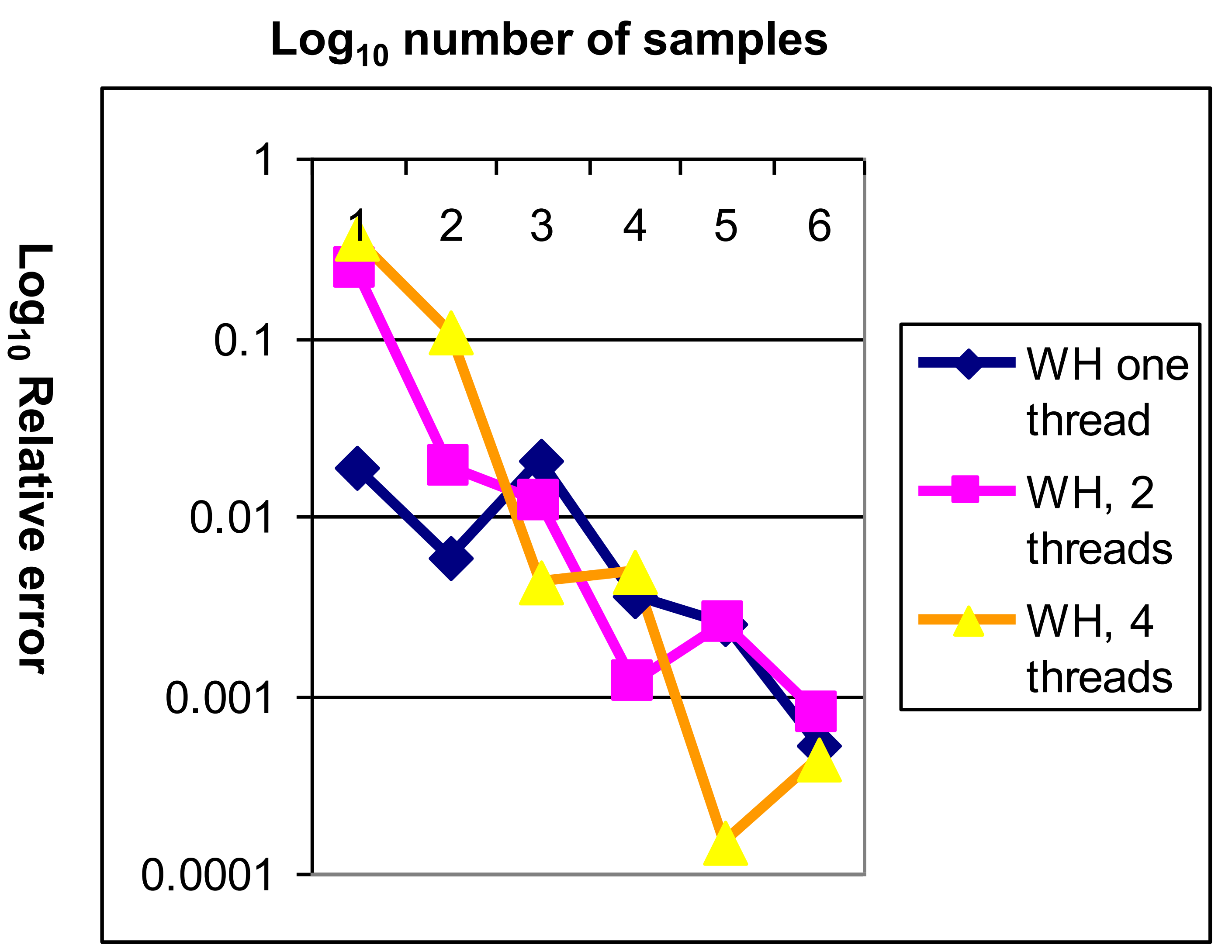
Delete the stream when you are done

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 i^{th} 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.

Challenge Problem Solutions

- Challenge 1: molecular dynamics
- Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- Challenge 6: Producer-consumer

Challenge 3: Matrix Multiplication

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?

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

Challenge Problem Solutions

- Challenge 1: molecular dynamics
- Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- → Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- Challenge 6: Producer-consumer

Challenge 4: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).

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

Challenge 4: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program two different ways
  1. Use OpenMP tasks

  ➡ 2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).

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

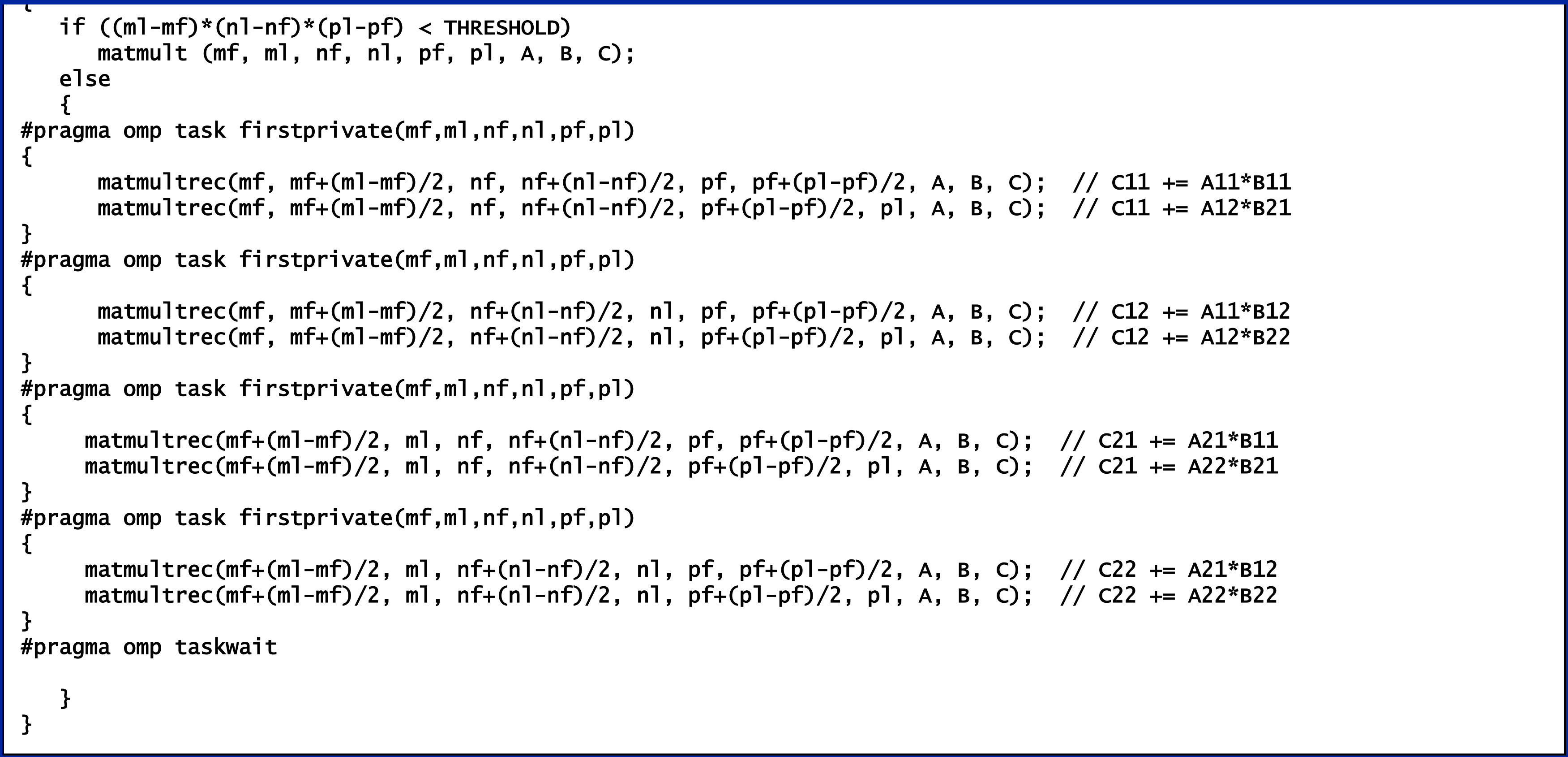
Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

Challenge Problem Solutions

- Challenge 1: molecular dynamics
- Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- Challenge 6: Producer-consumer

Recursive matrix multiplication

- 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 firstprivate(mf,ml,nf,nl,pf,pl)
{
      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 firstprivate(mf,ml,nf,nl,pf,pl)
{
      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 firstprivate(mf,ml,nf,nl,pf,pl)
{
     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 firstprivate(mf,ml,nf,nl,pf,pl)
{
     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

   }
}
```

Challenge Problem Solutions

- Challenge 1: molecular dynamics
- Challenge 2: Monte Carlo Pi and random numbers
- Challenge 3: Matrix multiplication
- Challenge 4: linked lists
- Challenge 5: Recursive matrix multiplication
- → Challenge 6: Producer-consumer

Example: 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 == 0){
                #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

The problem is this program technically has a race … on the store and later load of flag.

Atomics and synchronization

```
int main()
{   double *A, sum, runtime;
   int numthreads, flag = 0, flg_tmp;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
      #pragma omp section
       {  fill_rand(N, A);
          #pragma omp flush
          #pragma atomic write
                 flag = 1;
          #pragma omp flush (flag)
       }
       #pragma omp section
       {  while (1){
             #pragma omp flush(flag)
             #pragma omp atomic read
                  flg_tmp= flag;
              if (flg_tmp==1) break;
          }
          #pragma omp flush
          sum = Sum_array(N, A);
       }
     }
}
```

This program is truly race free … the reads and writes of flag are protected so the two threads can not conflict.